CONGESTION-RESILIENT TCP

BY

LENHARD W. WINTERROWD

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Adviser:

Professor David M. Nicol

ABSTRACT

In response to the advent of publicly-accessible high bandwidth links, we explore a potential resiliency enhancement to TCP that makes use of extra bandwidth to do forward error correction using the exclusive-or binary operation. We implement this XOR-Packets enhancement in the FreeBSD network stack and show that it guarantees the ability to reconstruct any single packet loss without the need for retransmission. Furthermore, in the unlikely best case, up to a third of the TCP stream can be lost without the need for retransmissions. We test our scheme against FreeBSD's default NewReno implementation with and without Selective Acknowledgment and find that our enhancement's goodput scales better than both for large stream sizes in high network congestion conditions.

To my parents, for their unwaivering love and support.

ACKNOWLEDGMENTS

Special thanks to my family for always believing in me. Thanks to Thomas Andrews and Lawrence Erickson for helping to keep me sane throughout this undertaking. Thanks to Joshua Cranmer for use of his second laptop in the physical testbed, especially on such short notice. Huge thanks and appreciation go out to Josh Hall for helping debug the testbed's network configuration with me and for confirming its validity on industrial routers. Finally, this thesis was possible in no small part due to the exceptional efforts and flexibility of my Adviser, Professor David Nicol, who went above and beyond the requirements of his position to provide feedback on my work and help make timely submission possible.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

As the global internet framework expands such that gigabit links are becoming available to the general public [1], we are seeing a new emerging problem-space defined by two questions. First, how can we make use of the increased bandwidths available? Second, how can we improve the reliability and general performance of the *existing* internet infrastructure to remain resilient aginst additional congestion in the presence of new bandwidth-intensive applications designed to utilize gigabit links?

While the Transmission Control Protocol over IP (TCP/IP) [2] [3] remains the dominant protocol for reliable internet communication, most "bandwidth-hungry" applications that can make use of gigabit bandwidths, such as high definition streaming video, rely instead on the User Datagram Protocol (UDP) [4] to achieve higher throughputs and lower latencies despite its best-effort behavior and lack of regard for network conditions. This leads us to question how TCP can evolve to better utilize gigabit links while retaining desirable performance characteristics in legacy networks.

The purpose of this thesis is to establish an enhancement to TCP that achieves improved resilience to the high network congestion levels induced by bandwidth-hungry applications by leveraging some additional bandwidth ourselves. We aim to add increased reliability to high-priority TCP streams by using some of the additional available bandwidth to allow limited packet loss without damaging effective protocol throughput or transmission latency.

Section 2 contains our problem definition and the generalized principles behind our solution. Section 3 establishes the details behind our protocol design. Section 4 explains the implementation details of our protocol. Section 5 describes our protocol testbed. Section 6 discusses the results of our tests and their implications. Section 7 suggests directions of future research.

CHAPTER 2

PROBLEM DEFINITION

We first present a review of relevant work, both in the problem-space of TCP enhancements and that of forward error correction. We then examine the barriers to congestion-resiliency inherent to TCP and finally propose a solution.

2.1 Literature Review

2.1.1 Overview of TCP Variants

TCP Tahoe [5] added the standard "slow-start" and "fast-retransmit" algorithms still used by many modern TCP implementations. Slow-start increases the sending window by one "Maximum Segment Size" (MSS) for every packet acknowledgement (ACK) received until a threshold or a TCP timeout. This generally results in the window size approximately doubling every round-trip time (RTT). After a set threshold, TCP switches to the "congestion-avoidance" state, in which it only increases the window size by one MSS per RTT. In the event of a timeout, the congestion window is reset to 1 RTT. "Fast-retransmit" treats three duplicate packet acknowledgements as a timeout and retransmits the unacknowledged packet.

In TCP Reno [6], the congestion window is only halved instead of set to one MSS on reception of three duplicate ACKs. Reno also adds "fast-recovery" in which it only retransmits the unacknowledged packet and waits for an ACK. If none arrives, the protocol times out, otherwise it returns to the "congestion-avoidance" state.

TCP NewReno [7], which forms the basis for our protocol enhancement, further enhances the fast-recovery state by attempting to keep the transmit window full. For every duplicate ACK that arrives, NewReno sends a new

unacknowledged packet from the end of the congestion window. If an ACK indicates progress, NewReno's timeout is reset. In this way, NewReno can fill in large sequential gaps in the congestion window at the rate of one packet per RTT without timing out. If the entire offending congestion window is ACK'd, NewReno returns to congestion-avoidance.

Selective Acknowledgements (SACKs) [8] are a header-option extension available to any TCP implementation that allow positive acknowledgement of discontinuous TCP segments, or "gaps" in the receive window. This allows for targeted retransmission of any missing segments, eliminating most unnecessary retransmissions. In practice, TCP-SACK can provide a notable performance increase over Tahoe and Reno for a lossy link [9].

TCP Westwood [10] enhances NewReno by inferring the link bandwidth via low-pass filtering of the ACK receive rate. It then uses this information to modify slow-start such that the TCP send rate scales up faster, providing increased performance on links with a high delay-bandwidth product. TCP Westwood+ [11] fixes inaccuracies in Westwood's bandwidth estimation that result from ACK compression.

TCP Vegas [12] introduces a class of congestion-control algorithms that estimate and use RTT to scale the TCP send rate. Other TCP variants based around RTT measurement include TCP NewVegas [13] and FAST TCP [14].

TCP Illinois [15] uses both packet loss and delay to estimate network congestion. TCP BIC [16], CUBIC [17], HSTCP [18], and H-TCP [19] all perform more aggressive window scaling and recovery to increase performance on high delay-bandwidth product links.

TCP-FIT [20] is designed to perform well in both lossy wireless and high delay-bandwidth product environments and can achieve significant performance improvements over many of the previously mentioned variants, particularly over Wifi or CDMA (3G) links.

2.1.2 TCP and Forward Error Correction

"Forward error correction" refers to any encoding scheme where redundant data is intentionally included into a stream of data for the purpose of reconstructing all or part of the original stream under error conditions. This is

most commonly seen in hardware in the form of error-correcting codes (ECC) or full device redundancy (ie a RAID 1 hard drive array). Schemes can be static or adaptive, where the amount of redundant data encoded changes depending on the level of reconstruction desired, often proportional to the expected error rate.

Link-layer FEC schemes such as [21] can provide resilience against per-bit link transmission errors but do little to prevent congestion-induced losses. In fact, even the IEEE 802.11g wireless standard [22] includes explicit support for forward error correction at the cost of about 32 Mbps of average throughput. [23] exhibits the behavior of TCP-SACK over a generic Automatic Repeat Request (ARQ) wireless scheme with FEC. However, the FEC is not built into the TCP protocol itself.

[24] simulates a generic datagram protocol with adaptive FEC. For simplicity, it is based on Asynchronous Transfer Mode (ATM), which uses fixed-sized cells as opposed to IP's variable-sized packets. In this generic protocol, k packets are encoded into n fixed-size cells with a level of redundancy h such that $n = k+h$, and receipt of any k out of n cells allows reconstruction of the original scheme. Potential encoding schemes are mentioned, but the choice of scheme is left abstract. The potential for reconstruction scales quite well as block-size n increases, but for Quality-of-Service dependent applications that require packet delivery within a set deadline, this imposes a direct tradeoff between minimum delay and the overhead from redundancy.

TCP Boston [25] [26] provides the closest basis for our modified TCP implementation. However, it is primarily concerned with leveraging redundancy to circumvent the overhead of the IP packet fragmentation that occurs for TCP/IP packets transmitted over ATM networks. Like [24], FEC is applied to ATM cells vs IP packets since the loss of any individual ATM cell corrupts the entire fragmented IP packet.

2.2 The TCP Retransmission Penalty

Due to the multiplicative-increase behavior of slow-start, most TCP variants almost guarantee packet loss before reaching the threshold to start congestion-avoidance, particularly in the presence of other network traffic. This property of TCP is especially problematic for latency-sensitive streams.

Even if TCP fails to deliver time-sensitive data within a required latency, TCP will continue to retransmit the potentially-stale data until it succeeds. This is the primary reason that video and VoIP (Voice-over-IP) streams use UDP, despite its tendency to *cause* congestion.

We now explore the degree to which a single packet drop actually affects the delivery time of the dropped packet. Assume a link latency of bandwidth *bandwidth* bits/s and latency *latency* ms with a current TCP send window size of at least 4 packets. In the best case, we consider a connection where Packet 9 has been ACK'd with $ACK(10)$ indicating the next expected sequence number. Now assume that Packet 10, with a size of *payload* bytes, is dropped, but packets 11, 12, and 13 still arrive at the receiver. This results in 3 duplicate $ACK(10)$s, which take at least

$$latency + 3\left(\frac{320}{bandwidth}\right) \tag{2.1}$$

seconds to propagate back to the sender (where 320 is the minimum TCP/IP header size in bits). The sender must then retransmit Packet 10 which takes

$$latency + \left(\frac{8\,(payload) + 320}{bandwidth}\right) \tag{2.2}$$

seconds to arrive at the sender, thus giving a total retransmission penalty of

$$latency + 3\left(\frac{320}{bandwidth}\right) + latency + \left(\frac{8\,(payload) + 320}{bandwidth}\right). \tag{2.3}$$

$$Penalty_{rxmit} \geq 2\,(latency) + \frac{4\,(320) + 8\,(payload)}{bandwidth}. \tag{2.4}$$

For large link latencies this can be a sizeable penalty, and this is the *best* case. We must also factor queuing delay into *latency*, which is generally function of congestion and could easily be quite high in this scenario, immediately after a packet drop.

2.3 TCP Packet Loss

TCP packet loss, despite widespread use as a measure of network congestion, is really only a *binary reactive* metric: *binary* because we either lose a packet or do not and *reactive* because we cannot measure loss or accurately predict it until we actually lose a packet. This is primarily why TCP variants like TCP Vegas [12] use RTT instead of packet loss to estimate congestion. Round-trip times can be measured such that we quickly have a fairly accurate expected value. Unfortunately, RTT is also a *preemptive* congestion mechanism. In the absence of actual packet loss, RTT-based protocols consistently predict congestion before loss-based protocols and consequently scale their congestion windows back "early" (as compared to loss-based protocols). When loss-based TCP variants are also present, this results in a self-induced "unfair" share of the contended link.

2.4 Lossy TCP Via XOR

Such behavior is a large part of the reason why packet loss is still such a popular congestion indication mechanism. Now, given the preferrential behavior of this *reactive* mechanism, what if we could remove the *binary* characteristic from packet loss? While it makes little sense to drop a fraction of a packet, we *can*, via FEC, modify TCP such that some packets can be lost without disrupting the stream as a whole.

We now consider a stream subset of at least N packets with $N \geq 2$, as adding redundancy for a single packet can only be achieved by full duplication and thus is trivially achieved with an overhead of 100%. The exclusive-or binary operator (XOR, represented as $\oplus$) provides a fast mechanism for creating a lossy environment where any 1 in $N+1$ packets may be lost. Given an average loss rate of $L \leq \frac{1}{3}$, we should select N such that

$$2 \leq N \leq \frac{1}{L} - 1. \tag{2.5}$$

Now given our N packets, we construct an XOR-packet such that

$$P_{XOR} = \bigoplus_{i=1}^{N} P_i \tag{2.6}$$

where P_i is the payload of packet i. We assume equal sized packets for the moment but will see in Section 2.4.2 that only the maximum-sized packet matters.

2.4.1 Optimality of Single-Packet Lossiness

Theorem 1. *For any N original packets and an added packet $P_{XOR} = \bigoplus_{i=1}^{N} P_i$, we can lose any one packet and still reconstruct that packet.*

Proof. The XOR operation is associative and commutative. Thus for $N > 2$

$$P_{XOR} = \bigoplus_{i=1}^{N} P_i = \left(\bigoplus_{i=1}^{k-1} P_i\right) \oplus P_k \oplus \left(\bigoplus_{j=k+1}^{N} P_j\right) \tag{2.7}$$

for arbitrary k without loss of generality. Now, for any two packets

$$(P_i \oplus P_j) \oplus P_j = P_i, \tag{2.8}$$

and thus our claim is trivially satisfied for $N = 2$ by the properties of XOR. Furthermore, by applying Equation 2.8 to Equation 2.7, we get

$$P_{XOR} \oplus \left(\bigoplus_{i=1}^{k-1} P_i\right) \oplus \left(\bigoplus_{j=k+1}^{N} P_j\right) = P_k. \tag{2.9}$$

Since k is entirely arbitrary, assume that we lose exactly one packet: P_k. Now by Equation 2.9, given the N other packets other than P_k, we can reconstruct P_k. □

Note that for a stream of X packets, as N decreases, it is possible, although extremely unlikely, to lose up to the limiting case of $\lfloor \frac{X}{N+1} \rfloor$ packets without requiring a retransmission. For a choice of $N = 2$, this equates to a potential for up to 33.3% packet loss. After this we *must* drop useful data.

2.4.2 Overhead of Single-Packet Lossiness

Assume that the maximum payload size of any packet to be XOR'd is M bytes. Also assume that we have a unique per-packet header of H bytes. Then for every N packets, we add an overhead of $(M + H)$ bytes in the form

of P_{XOR}. In the case where all packets are the same size, we thus have an overhead of

$$\frac{(M+H)}{(M+H)(N)} = \frac{1}{N}. \tag{2.10}$$

Now assume that all packets are *not* the same size. We can still create a valid P_{XOR} of size M bytes by padding payloads with zeros without affecting Theorem 1. However, our overhead does go up in the worst case (where 1 packet is of size M bytes and $N-1$ packets are of size of 1 byte). This worst case overhead is

$$\frac{(M+H)}{(M+H)+(N-1)(1+H)} \tag{2.11}$$

For the specific protocol designed and implementated in this paper (in Sections 3 and 4 respectivel), we select $N = 2$, as this allows to examine both the worse case overhead and best case lossiness. It is left to future work to explore an adaptive N based on the measured network packet loss rate for a persistent TCP connection. We also estimate $M = 1448$ bytes and $H = 52$ bytes. Plugging these values into Equation 2.10 gives us a best case overhead of

$$\frac{1}{2} = 50\% \tag{2.12}$$

and (via Equation 2.11) a worst overhead of

$$\frac{(1500)}{(1500)+(1)(53)} = \frac{1500}{1553} \approx 96.6\%. \tag{2.13}$$

2.4.3 Worst Case Loss Latency

In the worst case single-packet loss scenario, the maximum-sized *payload*-byte first packet of an N packet XOR group is lost for a link of bandwidth *bandwidth* bits/s. Given a header size of H bytes, that all other packets are also of maximum size *payload*, and that P_{XOR} is placed at the end of the stream, it will take

$$\frac{8N\,(payload+H)}{bandwidth} \tag{2.14}$$

seconds before the packet can be reconstructed. We ignore link latency in this case as the dropped packet essentially already paid that cost (most packets will be dropped or determined to be corrupted at a router or endhost).

CHAPTER 3

PROTOCOL DESIGN

To help mitigate the problems with TCP presented in Section 2, we attempt to modify TCP such that it is loss-tolerant. This is essentially achieved by transmitting additional redundant data out-of-band, or rather, not explicitly buffered within TCP's transmission window. Given two TCP packets, we create a third packet where this additional packet's payload is the XOR'd data of the previous two packets. This third packet, hereafter referred to as an "XOR-packet", is then sent along with the original two. The receiver's behavior depends on the arrival-order of the three packets, which can result in several cases. For additional clarity, we refer to the first original packet as "Packet 1" and the second original packet as "Packet 2".

Ideally, both Packet 1 and Packet 2 will arrive at the receiver. In this case, the loss or arrival of the XOR-packet is irrelevant. It can be lost safely or discarded if it arrives since all of the data it was formed from has already been received. Figure 3.1 shows the additional successful reconstruction cases for XOR-packets.

If an XOR-packet arrives after one of the original packets but before the other original packet, whether due to loss of the original packet or out-of-order arrival, we can completely reconstruct the missing packet. Suppose that Packet 2 and the XOR-packet have arrived, but Packet 1 has not. Packet 1's payload can be fully reconstructed by taking the XOR of the payloads of Packet 2 and the XOR-packet. Similarly, if Packet 2 is missing but Packet 1 and the XOR-packet have arrived, we can reconstruct Packet 2's payload. For the moment, we assume that Packets 1 and 2 are the same size. Combination and reconstruction of different-sized packets is discussed in Sections 3.2 and 3.3 respectively.

If only one of the three packets arrives, TCP's retransmission mechanism will apply as usual. If the only packet to arrive is the XOR-packet, the sender only needs to retransmit Packet 1 *or* Packet 2. Any selective acknowledge-

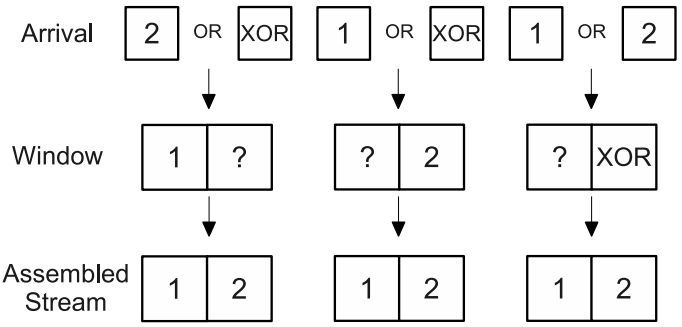

Figure 3.1: Successful arrival and subsequent reconstruction cases for XOR-packets.

ment scheme should be modified to reflect this. While XOR-packets themselves could feasibly be retransmitted, to maintain a lower overhead, we do not explore this possibility and only allow the retransmission of the original data. For our implementation, this results in the property that XOR-packets are never retransmitted.

3.1 Packet Selection

The scheme for selecting the two packets to XOR can be varied for different desired properties, performance characteristics, and ease of implementation. We look at some general global selection guidelines before highlighting the strengths and weaknesses of several such schemes.

3.1.1 Selection Guidelines

Control packets, those with the SYN, FIN, or RST TCP flags should never be XOR'd. These often require special processing which would divert them from the TCP reassembly queue. XOR'd control packets may also lead to problematic TCP behavior by entering the reassembly queue instead of being specially handled. Any zero length packets, including acknowledgement-only packets, should not be XOR'd as a zero size would lead to a fully-redundant

XOR with 100% overhead as opposed to an average of 50% (overheads will be discussed further in Section 3.4).

We also recommend that retransmitted packets not be XOR'd for two reasons. First, it may confuse some packet selection schemes with respect to which packets were XOR'd. Second and more importantly, a restransmitted packet means that some form of congestion likely occurred and we have already paid the TCP retransmission penalty, which XOR-packets are designed to avoid. We argue that XOR-packets are generally better used to progress the current window such that transmission can advance more significantly after the retransmission.

3.1.2 Simple Cache Selection

Every time a packet is sent, cache that packet. When the next packet is sent, if the cache is non-empty, after sending the current packet, XOR it with the cached packet. Send this newly created packet, then clear the cache. Any time a cached packet's receipt is acknowledged, clear the cache.

Ignoring additional latency due to our overhead, this scheme achieves theoretical minimum transfer latencies for any single packet loss or an evenly-distributed packet loss rate of up to 1/3. This avoids the TCP retransmission penalty referenced in Section 2.2. It is also fairly straightforward to implement.

For a fixed N, the number of packets to XOR as discussed in Section 2.4, Simple Cache Selection limits the extensibility of our protocol. For instance, we cannot safely cache any restransmits or create an XOR-packet from one or more retransmits since this scheme expects XOR-packets to be constructed of two in-order packets. While there could be substantial gains from allowing non-sequential packets to be cached, the logic to reconstruct a packet on arrival becomes significantly more complex. We also would need to be careful of duplicate retransmissions or the possibility of multiple XOR-packets with one or more shared source. As such, we choose to accept the limited extensibility of this scheme for our testing purposes.

Without the ability to cache non-sequential packets, the loss of any two packets in a group of three still requires a retransmit. Thus as packet loss becomes increasingly bursty as opposed to evenly distributed, a common

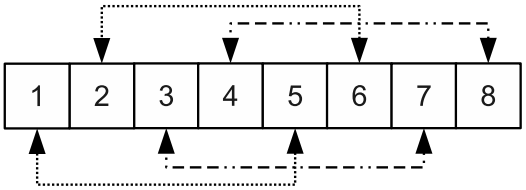

Figure 3.2: Packet selection using Spaced Window Selection

occurrence in real networks, the benefits of this scheme should diminish. As such, while we have a theoretical basis from which to expect weak congestion resilience, we must rely on the experimental results in Section 6 to determine its performance under fairly high network congestion.

3.1.3 Spaced Window Selection

Every time the send window is full and waiting on acknowledgements, take $offset = (sizeof(SEND\_WINDOW)/2)$ and for each packet from the oldest unacknowledged packet, XOR with the packet *offset* packets ahead of it in the window. We then send a burst of XOR-packets, as opposed to sending after every two. This scheme is shown in Figure 3.2.

By maximizing the spacing between the XOR'd packets, we add a degree of resiliency against consecutive packet loss. While the XOR burst itself is not resilient to consecutive packet loss, we also do not need to consider XOR-packets for retransmission unless desired. If retransmission of XOR-packets is undesirable, it is necessary to maintain a reference to the last sequence number XOR'd and exclude packets with the same or lower sequence numbers from the window.

Like Simple Cache Selection, this scheme achieves theoretical minimum transfer latencies for any single packet loss or an evenly-distributed packet loss rate of up to 1/3. However, this scheme also helps mitigate the bursty-loss problem mentioned in Section 3.1.2 both by XORing packets with non-incremental sequence numbers and by sending all of the extra redundant data after a burst of the original data. Now, if two subsequent original packets are lost, the stream may still be reconstructed as long as the two

necessary XOR-packets both arrive. As an additional benefit, this scheme only transmits data during a time when TCP would otherwise be idle waiting for an acknowledgement. While it still affects link utilization and indirectly affects throughput, the direct negative effect on throughput is reduced.

The additional resilience of this scheme gained through spacing comes at the cost of latency and complexity. If an original packet is lost in this scheme, the receiver must wait until the next XOR burst to reconstruct. In some cases, this may already be equivalent to a retransmission. Regarding complexity, most TCP implementations maintain byte buffers rather than packet buffers. For ideal functionality, packet boundaries would need to be established and maintained, increasing the TCP state stored, especially for large windows of small packets.

This scheme also provides very little benefit to steady-state TCP streams common to low-latency links where the window is constantly moving and rarely stalls to wait for an acknowledgement of its earliest byte. If the entire window is not in-transit, this mechanism will not trigger. Section 3.1.4 provides a modified scheme to help address this problem.

3.1.4 Spaced Subwindow Selection

Select an X-packet subspace of the current sending window (ie $X = 4$ of 8 packets). Maintain a reference to the most recent packet XOR'd and treat the most recent X packets as the entire window was in Spaced Window Selection (Section 3.1.3). If the earliest sent but unacknowledged packet (*snd_una*) is advanced past the most recent XOR reference, the XOR reference should be set to the new *snd_una*, even if it temporarily decreases the window size to less than X. Spaced Subwindow Selection is shown in Figure 3.3.

This scheme has the same basic advantages and disadvantages as Spaced Window Selection, but it behaves better with a steady-state sending window. In the event that even the X-packet subspace becomes steady-state, X may be left adjustable and potentially scaled with the size of the window or the measured RTT, depending on the TCP implementation. Spaced Subwindow scaling exceeds the scope this paper and is left to future work.

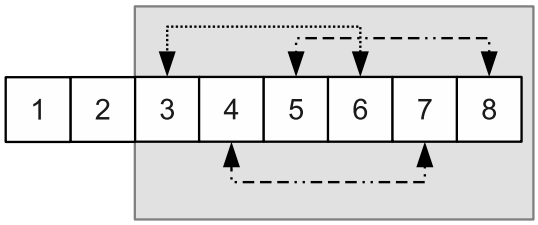

Figure 3.3: Packet selection using Spaced Subwindow Selection

3.2 Packet Combination

If the payloads of the two packets to be XOR'd are of different sizes, the difference is padded with zeros and thus, after the smaller payload, contains only the clear payload of the longer packet. The length of the shorter packet should be saved, as it will be written to the TCP header.

While the XOR operation itself is farily straightforward, there is some question about whether to XOR data only or the entire packet. While the most robust method of packet combination would include XORing the TCP headers as well as the data payload, the addition of another header could feasibly result in packets that exceed TCP's maximum packet size. As such, we look at performing the XOR operation for data only and combining the TCP headers of two packets. Again, for the rest of this paper, we assume the Simple Cache Selection XOR-packet selection scheme.

Simple Cache Selection and our exclusion of retransmitted packets implies that any XOR-packet will always directly follow the most recently transmitted packet. As such, we can begin from a copy of the most recent header to guarantee any timestamps in the copy are valid. Much of the header will already be correct as a result including the source port and destination port numbers.

The sequence number should always be the greater sequence number of the two XOR'd packets. Given Simple Cache Selection, this will always be the sequence number of the last sent packet and will not need to be changed.

Depending on the selection scheme, the acknowledgement number would either be redundant (a duplicate ACK, which may serve to trigger retransmissions in some TCP enhancements via fast-retransmit) or old, either of which may cause unexpected or undesired behavior on the receiver end. We

choose to replace the value in the "Acknowledgement Number" field with the payload size of the smaller XOR'd packet. We also propose a new "XOR" flag as one of the currently 'reserved' bits of the TCP header to indicate that this packet is an XOR and should be treated accordingly, including ignoring the "Acknowledgement Number" field.

By our general selection guidelines (Section 3.1.1), the most recently sent packet will never have the SYN, FIN, or RST TCP flags set, so we can ignore any flags, although it is safest to toggle any set flags but PSH to off.

The ACK flag must be turned off, as the "Acknowledgement Number" field is no longer valid. This may lead to incompatibilities with TCP implementations that assume every packet will have either the SYN flag or ACK flag set. The ACK flag is supposed to indicate that the given acknowledgement number is valid, and while it generally should be set, the acknowledgement field is *not* valid in this case, keeping our usage loosely within TCP specifications. While our new XOR bit primarily mirrors an ACK that is turned off, the XOR bit was added primarily for ease of implementation on systems always expecting the ACK flag.

The "Data Offset" and "Window Size" fields should not be modified. The "Urgent Pointer" field should generally not be used as it is recommended to set the URG flag to off. However, a valid XOR-packet extension (ie for Spaced Window Selection) may be to use the "Urgent Pointer" field to store the shorter XOR'd packet length instead of the "Acknowledgement Number" field. The "Acknowledgement Number" field could then be used to store the sequence number of the second XOR'd packet for easy reconstruction.

TCP options should generally be left as they are, although they may be removed if desired. Finally, the TCP checksum must be recomputed to reflect packet modifications.

3.3 Packet Reconstruction

On XOR-packet arrival, if its sequence number has not yet been acknowledged (ie at least one source packet has not yet arrived), then the XOR-packet should be added to the TCP reassembly queue. If one of the XOR-packet's source packets "S" is in the reassembly queue, the original packet should be reconstructed.

If S was the smaller packet, we just XOR S's payload onto the XOR-packet to achieve reconstruction. Otherwise, we discard the difference in packet sizes from the end of the reconstructed XOR-packet. Any receiving interface should return the XOR-packet's payload size to the TCP stack, thus allowing for easy computation of this difference (recall that the size of the smaller packet was stored within the "Acknowledgement Number" field).

Additionally, if the receipt of the reconstructed packet would have trigge an acknowledgement, reconstruction should trigger an acknowledgement as well. Otherwise, XOR-packets should not be acknowledged to avoid duplicate ACKs. This allows XOR-packets to still behave reasonably well with congestion control algorithms that rely on duplicate acknowledgements to indicate congestion and/or packet drops.

As an added note, when the next expected packet is received, socket buffers usually pass the receive data straight up to the user and free it from the receive buffer to allow for more receive space. Unfortunately, this behavior can make the first source packet ("Packet 1") of an XOR-packet inaccessible if the XOR-packet arrives after Packet 1. Thus Packet 1 must be cached or copied, depending on the OS implementation, before passing its data up to userspace. This concept is addressed again in our FreeBSD implementation details in Section 4.2.8.

3.4 Protocol Overhead

In the best case, where packets are equal-sizes, our XOR-packets scheme with Simple Cache Selection adds 50% data overhead (ignoring packet headers) to the network by sending an extra X bytes for every two packets, where X is the size of an individual packet. This overhead can be slightly mitigated by selectively sending XOR-packets or only sending during TCP idle times, such as when the send window is full and awaiting an acknowledgement, as mentioned in Section 3.1.3. However given any two packets, 50% is the minimum overhead for the third packet (as mentioned in Section 2.4.2).

In the worst case, we may have a 1 byte packet and a maximum-sized TCP packet, leading to just under 100% data overhead. However, with the commonality of MTU Discovery in modern TCP implementations [27], is it unlikely that this will be the average case. Even in interactive connections

that freely use the TCP PSH flag, it is highly unlikely that every other packet will alternate between 1 byte and the maximum TCP packet size.

CHAPTER 4

IMPLEMENTATION

For testing purposes, we modified the TCP implementations of both S3FNet [28] [29], a parallel network simulator, and FreeBSD [30], a well-known open-source derivation of BSD UNIX, to support XOR-packet generation and reconstruction.

4.1 S3FNet

Our initial implementation was added to the S3fNet parallel network simulator in an effort to develop a proof-of-concept. We were able to leverage the protocol structures within the simulator, such as the simulated abstraction of a TCP packet, to greatly simplify our implementation. While fairly extensive modifications to the TCP implementation were still necessary, particularly with respect to the TCP reassembly queue, we were able to implement a higher-level abstraction for XOR-packet classification and reconstruction that more closely paralleled Section 3's concepts than Section 4.2.8's complicated conditional tree. The logging already present for TCP was also extended to identify XOR-packets and record packet losses.

4.2 FreeBSD

We modified the FreeBSD network stack to support sending and receiving XOR-packets with the initial goal of verifying simulation results. We have since chosen to use FreeBSD virtual machines in a physical testbed for the bulk of our experimentation. Despite the apparent simplicity of the XOR-packets concept, even implementing Simple Cache Selection proved non-trivial. The following subsections detail the necessary modifications to the FreeBSD kernel as well as some essential background information on the OS

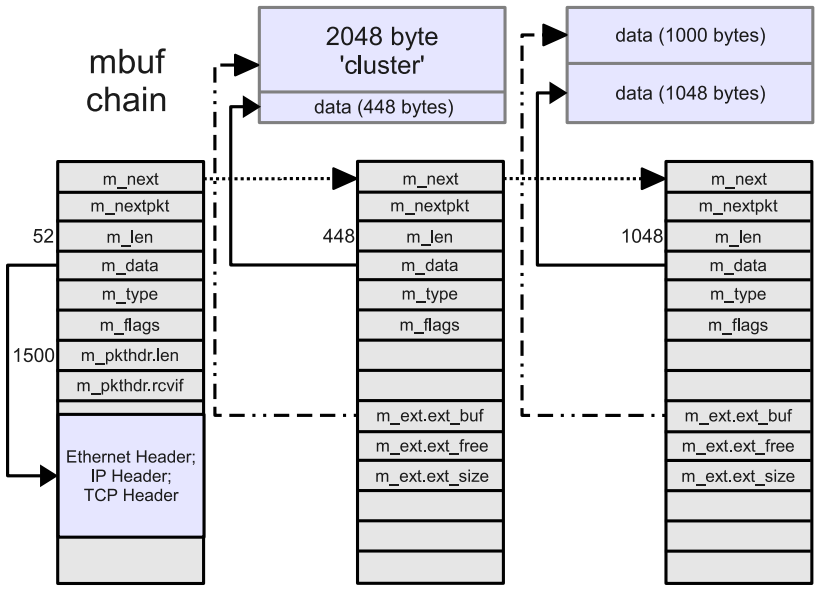

Figure 4.1: 1500 byte TCP packet in FreeBSD mbuf chain

itself and its network stack. For a more thorough background, we suggest referring to Wright & Stevens' TCP/IP Illustrated Volume 2 [31].

4.2.1 The FreeBSD Kernel's Dynamic Memory Model

In order to provide fast, efficient support for packet operations, dynamic memory management within the FreeBSD kernel's network stack works differently from that in conventional C code. All operations are performed on data structures called "mbufs", which are effectively preallocated 128 byte buffers that function mainly as linked-lists and are managed by the kernel. There are 4 separate types of mbufs: "data-only", "packet-header", "extended-data", and "extended-packet-header". Type is determined by the combination of two flags in the *m_flags* field: M_PKTHDR and M_EXT. On 32-bit machines, the headers corresponding to each of these 4 types range from 20 to 40 bytes.

The "packet-header" mbuf type adds access to a length field and a reference to the network interface on which any received packet arrived. If additional space is needed beyond the 100 remaining bytes of a packet header, the *m_next* field will point to another mbuf. Generally the *m_data* field of any

mbuf type points to the first byte of data stored within that mbuf, but it can point to an outside structure in one case. If the M_EXT flag is set, then the mbuf's *m_data* field references a "cluster", an extended memory block of 1024 or 2048 bytes (platform dependent). On our FreeBSD platforms, a cluster is 2048 bytes and thus is sufficent to hold a TCP packet of maximum size (1500 bytes). In the case of a 1500 byte TCP packet, such as that in Figure 4.1, we will generally have a linked list of two or more mbufs with the second's *m_data* field referencing a cluster.

Memory allocation returns an mbuf and optionally can wait for one to become available if all are occupied. Freeing returns possession of that mbuf to the kernel. Cluster allocation is done similarly, but clusters must be specially attached to an mbuf since they are reference counted to allow efficient sharing of data across multiple mbufs (ie for a socket buffer).

4.2.2 Adding an XOR Memory Primitive

Due to the reference counting of clusters (which store, among other data, our socket send buffers), we cannot create an XOR-packet without modifying the original send buffer, potentially impacting retransmissions. The last sent packet must be duplicated, including potentially allocating a new cluster before the XOR operation. Additionally, since we must work with chains of mbufs instead of regular byte buffers, we require a new memory primitive just to perform a simple XOR operation.

The *m_xor*() function, presented in pseudocode as Algorithm 1, adds XOR functionality to mbuf chains of arbitrary length and optionally containing one or more clusters. *len* bytes are XOR'd from the mbuf *src* onto the mbuf *dst* with the given data offsets. If *dst* runs out of space, a new mbuf is allocated with a cluster attached to it and the remaining bytes from *src* are *copied* into *dst* instead of XOR'd since the end result is the same as zeroing then xoring. This also avoids any case where the initial value of *dst*'s extended data may be uninitialized. After all data (*len* bytes) has been processed, the primitive returns the amount of bytes *dst* was extended by, if any.

Algorithm 1 XOR *len* bytes from *soff* into *src* to *doff* into *dst*

```
while soff needs seeking do
  decrement soff by src.m_len
  advance src to src.m_next
end while
while doff needs seeking do
  decrement doff by dst.m_len
  advance dst to dst.m_next
end while
set extend to 0
set mode to MODE_XOR
while not all bytes xor'd do
  if mode is MODE_XOR then
    xor min(src.m_len,dst.m_len,len) bytes from src onto dst
  else
    copy min(src.m_len,dst.m_len,len) bytes from src to dst
  end if
  decrement len by min(src.m_len,dst.m_len,len) bytes
  if src empty then
    advance src to src.m_next
  end if
  if dst empty then
    if at end of dst then
      allocate new mbuf and cluster to dst.m_next
      set mode to MODE_COPY
      add min(len,2048) to extend
    end if
    advance dst to dst.m_next
  end if
end while
return extend
```

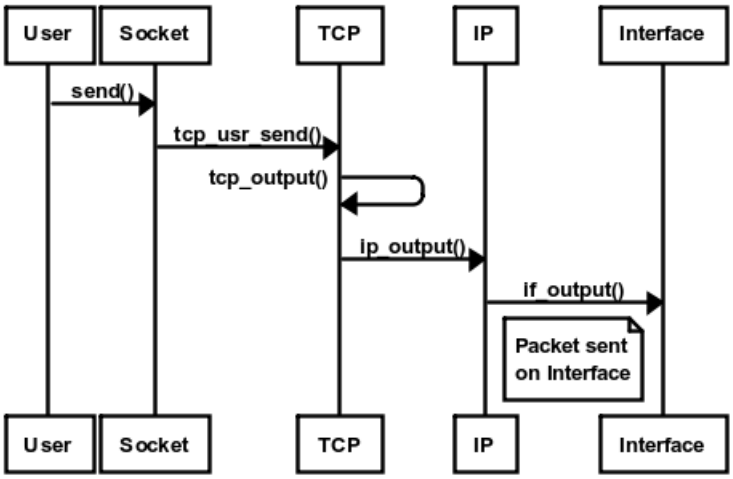

Figure 4.2: TCP callflow for a sent data packet

4.2.3 Overview of the FreeBSD TCP/IP Stack

The 484 byte *tcpcb* structure contains all of the per-connection information for TCP including window information and even the TCP reassembly queue. This structure is passed between all of the TCP-layer functions in the BSD network stack. Our extensions for caching XOR-packets add 36 bytes to its size (although this could feasibly be decreased) for a total structure size of 520 bytes per connection.

As of FreeBSD 9, support for Modular TCP Congestion Control [32] has been integrated into the network stack. This provides a relatively simple framework for developing and testing new congestion control schemes. Regrettably, due to the memory operations required for XOR-packets and the enhancement's goal of congestion-resilience vs congestion control, we could not leverage this mechanism alone for our work. At the very least though, it is possible to quickly switch the congestion control algorithm used for TCP between any number of provided implementations. Implementations include NewReno [7] (the system default), Vegas [12], CUBIC [17], H-TCP [19], "Hamilton-Delay" [33], and "CAIA Hamilton-Delay" [34].

Figure 4.2 shows the simplified callflow for any generic data packet sent from userspace. When a TCP socket sends data, it is passed to the socket layer where it is processed as a user send request. From there, it is passed to the TCP layer, where the data is appended to the socket buffer via

sbappendstream() and then fed into *tcp_output*(), the main processing function for sending TCP packets. There the header is computed and the packet is passed to the IP layer via *ip_output*(). After the IP header is populated, the packet is pushed to the network interface via *if_output*() and sent.

Figure 4.3 shows the simplified callflow for a generic TCP data packet received on some network interface. Packet reception causes an interrupt that results in *if_input*() which in turn calls *ip_input*() for TCP/IP packets. *ip_input*() checks the packet for consistency, checks if the packet should be forwarded, and reassembles segmented IP packets if necessary. Any TCP packet is then passed up to *tcp_input*() where the TCP header is verified and the connection control block (*tcpcb* structure) is identified. Most segments are then passed to *tcp_do_segment*(), where they undergo a check called "Header Prediction" [35]. Header prediction allows fast-track processing if the received packet is the next expected packet in the TCP stream and there is no outstanding reassembly queue. If prediction fails (ie for an out-of-order packet or a previously missing packet), *tcp_reass*() is called to update the reassembly queue. In both cases, if there is data to pass up to the Socket layer after processing, the socket receive buffer is updated and woken up to indicate the arrival of data to userspace. After this step, if an acknowledgement is required for the received data, *tcp_output*() is called to generate the ACK packet.

Our XOR-packets extension requires fairly comprehensive modifications to *tcp_output*(), *tcp_do_segment*(), and *tcp_reass*(). While *tcp_reass*() is contained within *tcp_do_segment*(), it required the most notable changes and thus will be examined on its own instead of with *tcp_do_segment*() as equal parts of *tcp_input*().

4.2.4 The *tcp_output*() Function

The bulk of the logic changes in *tcp_output*() revolve around caching packets to XOR and the selection guidelines established in Sections 3.1.1 and 3.1.2. We wish to consider only packets of non-zero length without control flags set that are not restransmissions. The TCP flags SYN, FIN, and RST will only be set on packets of zero length in FreeBSD's tcp_output(), allowing us to filter all but retransmitted packets with a single check. Retransmissions can be

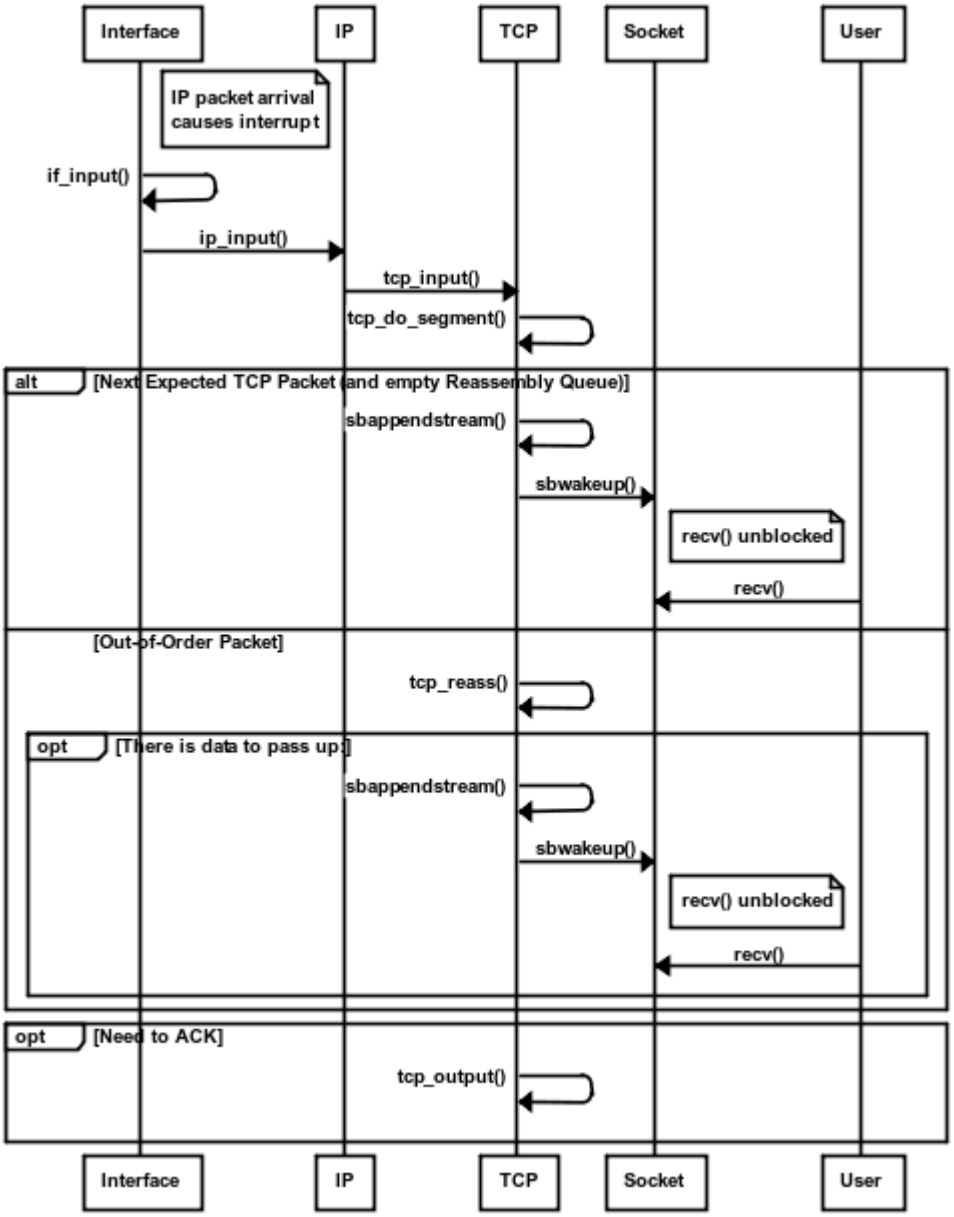

Figure 4.3: TCP callflow for a received data packet

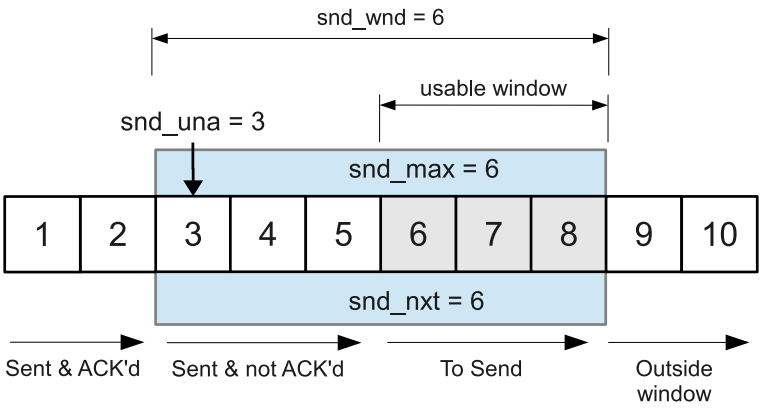

Figure 4.4: Depiction of the FreeBSD TCP send window and some relevant variables from the *tcpcb* structure

checked for by comparing the *tcpcb* structure's *snd_nxt* parameter, the next packet to be sent, to its *snd_max* parameter, the highest sequence number to be sent. If $snd\_nxt < snd\_max$, the current packet is a retransmission. Figure 4.4 shows a TCP window example in more detail.

Unfortunately, FreeBSD advances *snd_nxt* and *snd_max* before actually sending the packet. This effectively prevents us from checking if any already sent packet was a retransmit and thus requires us to maintain additional state about the current packet's retransmit status by performing checking if $snd\_nxt < snd\_max$ much earlier in *tcp_output*(). Failure to do so can result in retransmitted packets being XOR'd which can violate some simplifying assumptions for packet sequence numbers in *tcp_reass*().

After packet transmission, if the sent packet contained some payload and was not a retransmission, we "fake-cache" it by storing its sequence number and length, allowing reference into the socket send buffer but avoiding a wasted mbuf. We then set the *xor_valid* flag within the *tcpcb* structure to indicate that the next valid packet should generate an XOR-packet.

Immediately before any packet is pushed to the IP layer to be sent (via *ip_output*()), we check the *xor_valid* flag. If it is set, we physically cache it by fully duplicating the packet's mbuf, including any clusters, since *if_output*() will free the packet's mbuf chain upon sending. This copy is briefly stored within the *tcpcb* structure. If the current packet is sent successfully, the same code block that sets the *xor_valid* flag will notice a physically-cached packet and call *tcp_xor_send*(), detailed in Section 4.2.5.

Additionally, before any packet is sent in *tcp_output*(), we add a safety case where we reset the *xor_valid* flag if the sequence number of our fake-cached packet has already been acknowledged, thus causing the current packet to once again be fake-cached instead of physically cached, preventing output of an XOR-packet composed partially of old, already-received data.

4.2.5 The *tcp_xor_send*() Function

Algorithm 2 Generate XOR-packet from "fake-cached" packet FC and packet copy P

turn off ACK flag in $P.hdr$
set XOR flag in $P.hdr$
set $tcphdr.ackno$ to min($FC.length$,$P.length$)
set $doff$ to $P.hdrlen$
set $soff$ to $FC.seqno - snd\_una$
m_xor($so.snd\_buf$,$soff$,P,$doff$,$FC.length$) {see Algorithm 1}
recompute $P.hdr.checksum$
ip_output(P)
set $tcpcb.xor\_valid$ to **false**
set FC to $NULL$

tcp_xor_send(), presented in pseudocode as Algorithm 2, is the function added to the FreeBSD network stack to create and send XOR-packets. Starting from the physically-cached packet, we first turn off the ACK flag and set our XOR flag in the TCP header. We then update the acknowledgement number to be the smaller of the two packet sizes (trivial since the fake-cached packet's length is stored in the *tcpcb* structure) and compute the offset of actual data in the physically-cached packet (ie skip over the header). The socket send buffer is used as the mbuf for the second packet with $offset = last\_pkt\_seqno - snd\_una$, where *last_pkt_seqno* is the stored sequence number of the fake-cached packet.

The *m_xor*() function (see Section 4.2.2) is then called to generate the new payload. After computing the new TCP checksum, the XOR-packet is sent via *ip_output*() and consequently freed by *if_output*(). Finally, we reset the *xor_valid* flag to off, reset the now-freed physically-cached packet reference to NULL and return to normal *tcp_output*() execution.

4.2.6 Most Recent Segment Caching

Most network stacks understandably attempt to pass arriving data to the user as quickly as possible then free it from the socket buffer. FreeBSD is no exception; consequently, we must save a copy of the most recent expected packet, as noted at the end of Section 3.3, so that reconstruction can be still peformed for any arbitrary arriving XOR-packet. Unlike when we generate XOR-packets, this cached copy does not need to fully duplicate the current packet's mbuf chain. We can instead allow the kernel to reference count any referenced clusters since we will not be modifying the data of any regular received data packet. Obviously this also implies that XOR-packets will never be cached this way, but in fact, this is already achieved by diverting all XOR-packets towards the *tcp_reass*() function, discussed in Section 4.2.8.

In our implementation we actually present the *copy* to the socket receive buffer instead of the original. This is due to a behavior referred to in the FreeBSD source code's comments as a "delayed header drop". Once a reference to the TCP header is saved in a local variable, the packet's mbuf is seeked to reference the first byte of actual data via a call to *m_adj*(). The header itself is not freed until the mbuf is freed, but it is difficult to create any more references to the header without seeking the mbuf back, which generally requires information *in* the "dropped" TCP header. Since the socket receive buffer only cares about the actual data, we cache both the local header reference and the original packet itself and present our packet copy (copied *after* the "delayed header drop") to the socket buffer, thus avoiding the header problem entirely.

4.2.7 The *tcp_do_segment*() Function

tcp_do_segment() handles the bulk of packet processing for *tcp_input*(). To properly receive XOR-packets, we first must slightly modify the "Header Prediction" [35] logic. While XOR-packets are naturally excluded due to their unset ACK flag, it is safer to explicitly exclude any packet with the XOR bit set to guarantee that XOR-packets are always forced to *tcp_reass*(), discussed in Section 4.2.8. For any (non-XOR) packets that pass Header Prediction, we must cache a copy within the *tcpcb* struct as noted in Section 4.2.6.

Once forced to general packet processing, XOR-packets will undergo a test for duplicate data where $bytestodrop = rcv\_nxt - pkt.seqno$. Here rcv_nxt represents the next expected sequence number as stored in the *tcpcp* structure and $pkt.seqno$ is the received packet's sequence number. If $bytestodrop > pkt.length$, then the entire packet's data has already been received and we can drop it, even if it is an XOR-packet. However, this will still spawn an acknowledgement, which we *do not* want for XOR-packets. As mentioned in Section 3.3, we wish to avoid duplicate acknowledgements to optimize behavior with various congestion control algorithms.

After checking for duplicate data, *tcp_do_segment*() checks if any of the bytes in the current packet are beyond the end of the sequence window and drops them. Again XOR-packets require special processing as their length is the length of the longer source packet. If the *first* source packet is the longer of the two, the XOR-packet may not have any data beyond the window. To fix this, we use the length of the shorter packet (stored in the "Acknowledgement Number" field) for this comparison instead of the full XOR-packet length. Once again, we must suppress any acknowledgements of XOR-packets if this code block is entered.

Before acknowledgement processing, the FreeBSD tcp implementation drops all packets that do not have either the SYN or ACK flags set, which catches XOR-packets. This check was modified so that, like SYN packets, XOR-packets skip acknowledgement processing and reach data processing. At the data processing block, we tweak the logic such that XOR-packets are always forced towards the *tcp_reass*() function. Any packets that did not satisfy header prediction but still had the expected sequence number must be cached here as noted in Section 4.2.6 since they will avoid *tcp_reass*().

4.2.8 The *tcp_reass*() Function

In the original FreeBSD TCP implementation, every packet that makes it to *tcp_reass*() should be inserted into the TCP reassembly queue. With the addition of XOR-packets, we add the ability to discard packets and free their mbufs as needed instead of adding them to the queue. For instance, if there is ever an error reconstructing a packet, we discard the XOR-packet involved as it may have been modified in the process.

After some initial checks, such as limiting the size of the reassembly queue, *tcp_reass*() finds a queued segment that begins after the current segment. We extend this to also save the segment immediately preceeding the current segment (ie with same or lesser sequence number). Our processing now takes the form of case-handling the cases presented in Section 3. For clarity, a simplified version of our case handling is presented in Algorithm 3.

Generally, if we received an XOR-packet and either the packet before it in the queue or our last cached packet (from Section 4.2.6) was one of its source packets, reconstruct. If we received a normal packet and either the packet before it or after it in the queue is an XOR-packet constructed from this packet, reconstruct. However, line 5 is a special case. If we have a gap in the sequence number space and both source packets arrived after this gap, we could have both source packets for an XOR-packet in the queue, but the XOR-packet will not be classified as redundant data by *tcp_do_segment*(). In this case, we should just discard the XOR-packet (line 10).

If line 3 returns false, the unreconstructed XOR-packet will get added to the queue since we can never have duplicate XOR-packets; any other XOR must be unique from this one. If lines 4, 15, or 16 return false, then neither source packet is in the queue, and again, the unreconstructed XOR-packet will get added to the queue. A normal packet will always be added to the reassembly queue regardless of reconstruction.

For actual reassembly queue insertion, unreconstructed XOR-packets skip *tcp_reass*()'s check for redundant data. Furthermore, even if an XOR-packet's sequence number is the next expected sequence number, if it has not been reconstructed, we must prevent it from being added to the socket buffer. Finally, when regular data is actually presented, the latest packet added to the socket buffer must be cached to guarantee reconstruction (again, as in Section 4.2.6).

Normally, *tcp_reass*() returns either 0 for normal behavior or the FIN flag to indicate that a packet with the FIN flag set was in the reassembly queue. We modify this behavior to also return the ACK flag if an acknowledgement is needed. The ACK flag is only not returned if the processed packet was an XOR-packet for which reconstruction was not performed.

Algorithm 3 XOR Reconstruction Case-Handling, given arriving packet P and references P.next, P.prev, and P.prev.prev in the TCP reassembly queue

```
1: if P is an XOR-packet then
2:   if P.prev exists then
3:     if P.prev is NOT an XOR-packet then
4:       if P.prev was a source packet of P then
5:         if P.prev.prev is NOT the other source packet of P then
6:           tcp_xor_reconstruct(P.prev,P)
7:           set P.next to P.prev
8:           set P.prev to P.prev.prev
9:         else
10:          discard P
11:        end if
12:      end if
13:    end if
14:  else
15:    if a received packet CRP is cached then
16:      if CRP is the first source packet of P then
17:        tcp_xor_reconstruct(CRP,P)
18:        free CRP
19:      end if
20:    end if
21:  end if
22: else
23:  if P.prev is an XOR-packet then
24:    if P is a source packet of P.prev then
25:      tcp_xor_reconstruct(P,P.prev)
26:    end if
27:  end if
28:  if P.next is an XOR-packet then
29:    if P is a source packet of P.next then
30:      tcp_xor_reconstruct(P,P.next)
31:    end if
32:  end if
33: end if
```

4.2.9 The *tcp_xor_reconstruct*() Function

Algorithm 4 Reconstruct a source packet from an XOR-packet *XP* and the other source packet *SP*

```
m_xor(SP,0,XP,0,SP.len) {see Algorithm 1}
if SP.length ≥ XP.length then
  set XP.length to XP.ackno
  set size to XP.length
  set curmbuf to XP
  while curmbuf ≠ NULL do
    if size > 0 then
      if size < curmbuf.m_len then
        set curmbuf.m_len to size
      end if
      decrement size by curmbuf.m_len
    else
      set curmbuf.m_len to 0
    end if
    set curmbuf to curmbuf.m_next
  end while
end if
if reconstructed earlier source packet then
  decrement XP.seqno by XP.length
end if
reset XOR flag to off
set ACK flag
set XP.ackno to SP.ackno
```

tcp_xor_send(), presented in pseudocode as Algorithm 4, is the function added to the FreeBSD network stack to reconstruct data from an XOR-packet and one of its source packets. The packets XOR'd in *tcp_reass*() have already been seeked past their headers through the "delayed header drop" referenced in Section 4.2.6. As such, we can XOR them immediately. If the reconstructed packet is the smaller one, we then need to iterate through its mbuf chain and make sure lengths are set correctly. If the packet reconstructed was the earlier of the two source packets (the one with the lower sequence number), we should also adjust the reconstructed packet's sequence number. Finally, we turn the XOR flag off and, while it is mostly unnecessary, turn the ACK flag back on and copy the ACK value from the other source packet in case any code refers to the ACK field. While setting the ACK bit and "Acknowledgement Number" field is mostly unnecessary, it is completely safe

to do so.

4.2.10 FreeBSD's NAT Checksum Optimization

While the aforementioned changes to FreeBSD's TCP stack should be sufficient to support our XOR-packets enhancement, testing revealed that although it was getting set sender-side, the XOR bit was getting cleared before it reached a receiver's TCP layer. As of RFC 3540 [36] the "NS" bit, used for ECN-nonce concealment protection, was proposed for use in the TCP header and utilized one of the "Reserved" bits in the header's "Offset" field. This would correspond to a mask of "0x01" in the *th_x2* field of FreeBSD's *tcphdr* structure. As such, our XOR bit was given a mask value of "0x02" to make use of the next "Reserved" bit. However, FreeBSD surprisingly does not support the NS bit. We discovered that the *th_x2* field, while properly noted as "Reserved", is actually used for a Network Address Translation optimization within FreeBSD's libalias (NAT) library.

Network Address Translation (or NAT) [37] by nature must modify TCP packets, particularly source and destination addresses. However, each time the TCP Header is modified, the checksum must be recomputed. To assist in this, most network adapters have explicit hardware support for checksum computation. Passing off a packet for fast hardware-based checksum computation is a process known as "checksum offloading". FreeBSD's libalias has no knowledge of network adapters, as it was never meant to run in the kernel, and thus is not aware of checksum offloading. It is instead forced to perform the checksums in software, which can hinder performance significantly. Futhermore, it often only has partial "pseudo-headers" for locally-generated packets, so it *cannot* compute a correct checksum in some cases.

To avoid this problem, FreeBSD uses the TCP header's entire *th_x2* field as a single boolean field to indicate that a checksum needs to be performed. When a marked packet is received by offloading-aware code farther down the network stack, the checksum is computed and the field is cleared. Incidentally, this clears our XOR bit and would have also cleared the NS bit, had it been implemented.

As a fix, we designated the highest reserved bit (given mask "0x08") of the *th_x2* field to serve the same purpose that the entire field was previously being

used for. This required modification of six additional files in the FreeBSD networking stack and several other files across the OS that were aware of this optimization. However, this *did* solve the problem and XOR-packets were able to be successfully received.

4.2.11 Network Tool Modifications

In order to record any statistics pertaining to XOR-packets or our general modifications to the FreeBSD TCP implementation, it was necessary to modify FreeBSD's version of several network statistic utilities. First, we modified netstat, which provides hooks to the OS statistical structures, generally maintained per protocol. We added global counts of XOR-packets sent, XOR-packets received, XOR reconstruction candidates (XOR-packets that reach *tcp_reass*()), XOR reconstructions performed, and XOR reconstructions failed. These parameters were then all added to the output of the netstat binary.

We also modified the tcpdump [38] utility to display the XOR flag and show the "Acknowledgement Number" field even in the absence of an ACK flag. If ACK is not set, the field is displayed as "xorlen" instead. Incidentally, we discovered a bug in the tcpdump implementation during modification. TCP dump's generic formatting function to print packet flags expects that every packet will always have at least one flag set. For TCP, this is conventionally true, with either the SYN or ACK flag always set. With our implementation, we may have only the XOR flag set, but it is located in the *th_x2* field as opposed to *flags*, thus TCP flags appear empty. In this case, tcpdump prints out the last set of non-zero flags encountered. This was the result of an early exit from the flag formatting function and has been fixed for our usage.

CHAPTER 5

EXPERIMENTAL SETUP

Tests in both simulation and on our physical testbed were run for networks of comparable topologies that focused on congesting a bottleneck link. Both implementations contained the same actors in the form of a TCP client, TCP Server, UDP Flooder, and UDP Listener.

5.1 Simulation Methodology

Our simulation network was originally configured primarily as a proof-of-concept for an extremely congested link. Our chosen topology was a standard bottleneck link as depicted in Figure 5.1. All links were assigned equal latency, with latency varied from 100 μs to 1.0 s by powers of 10 as separate test cases. Links from the hosts to each router were established at 1 Mbps while the bottleneck link between routers was set as only 800 Kbps.

5.1.1 Reproducing our Physical Testbed

Reconstructing our physical network precisely in simulation proved difficult due mainly unknown parameters in our hardware, such as the router buffer size. We did not attempt to represent the virtual machine bridged links, instead evenly distributing our end-to-end latency across the three links traversed from Client to Server and from Flooder to Listener. Overall, we both reconstructed the topology of our physical testbed and attempted to match all latencies as closely as possible to those calculated in Table 5.3.

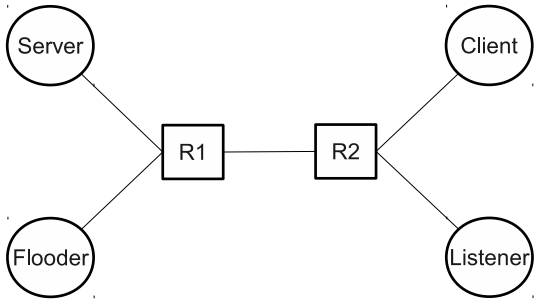

Figure 5.1: Simple Bottleneck Topology of Simulation Network

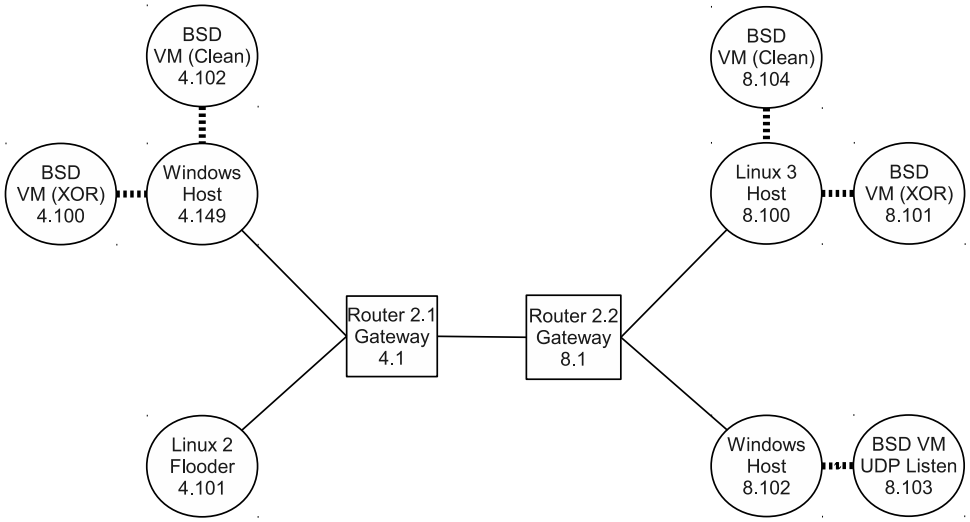

Figure 5.2: Topology of Physical Test Network

IP	OS	CPU Freq	Cores	32/64-bit	RAM	Role
4.100	BSD 9 (XOR)	3.20 GHz	1	32-bit	512 MB	VM Server
4.101	Linux 2.6	2.00 GHz	1	32-bit	2GB	UDP Flooder
4.102	BSD 9	3.20 GHz	1	32-bit	512 MB	VM Server
4.149	Windows 7	3.20 GHz	4	64-bit	6 GB	Host
8.100	Linux 3.2	2.93 GHz	4	64-bit	12 GB	Host
8.101	BSD 9 (XOR)	2.93 GHz	1	32-bit	512 MB	VM Client
8.102	Windows 7	1.73 GHz	4	64-bit	4 GB	Host
8.103	BSD 9	1.73 GHz	1	32-bit	512 MB	VM UDP Receiver
8.104	BSD 9	2.93 GHz	1	32-bit	512 MB	VM Client

Table 5.1: Table of Machine Specifications and Roles

5.2 Physical Network Methodology

The physical test network was implemented as a bottleneck link over 100Mbps ethernet between two identical Linksys E1000 consumer routers with wireless, NAT, and firewalls all disabled. The test network topology, including IP addresses on an isolated 192.168.0.0/16 subnet, is depicted in Figure 5.2. Each router is connected to two physical hosts, each of which may run one or more 32-bit FreeBSD virtual machines via VirtualBox [39]. On the 192.168.4/8 subnet, there is a Debian Linux host (our UDP flooder) running the Linux 2.6 kernel and a Windows host running two BSD virtual machines (our servers): one with our XOR-packet modifications and one without. On the 192.168.8/8 subnet, there is a Windows host running a single BSD virtual machine (our UDP listener) and a Debian Linux host running two additional BSD virtual machines (our clients): again, one with our XOR-packet modifications and one without. The roles and capabilities of all machines are detailed in Table 5.1. Additionally, all hosts have gigabit ethernet interfaces, and the hosts for the clients and servers have network interfaces of identical make and model, although firmware may differ slightly. The eth0 adapter on VirtualBox hosts is bridged with the host adapter. All firewalls and NATs are disabled.

5.2.1 Test Network Latencies

To estimate network latencies, "ping" was used to gather at least 100 latency samples of 24 point-to-point paths one at a time in an otherwise clear network. It was determined that pings originating from the routers were skewed

Scenario	IP Path	Min RTT (ms)	Avg RTT (ms)	StdDev
Client → Server	8.104 → 4.102	1.0570	1.30439	0.09662
Client → Listener	4.101 → 8.103	1.1300	1.39176	0.09389
VM → VM (same host)	8.104 → 8.101	0.4630	0.65766	0.12723
VM → VM (same subnet)	8.104 → 8.103	0.7560	0.98373	0.10486
VM Self-Ping (Windows)	4.102 → 4.102	0.1110	0.16695	0.01846
VM Self-Ping (Linux)	8.104 → 8.104	0.4630	0.66350	0.14669
Linux 3.2 Self-Ping	8.100 → 8.100	0.0270	0.03202	0.00143
Linux 2.6 Self-Ping	4.101 → 4.101	0.0130	0.02224	0.00318

Table 5.2: Selected Test Network Latencies

high, likely due to the specialized nature of the router processors and their low *general* computational ability. A list of notable minimum and average ping measurements are listed in Table 5.2.

We consider link latency, router forwarding latency, network stack latencies, and VM bridging latencies to approximate delays in our test network. Network stack latencies are considered per OS, ignoring the added overhead of our XOR implementation (BSD 9, Linux 2.6, and Linux 3.2). VM bridging latencies are considered per host OS (Linux 3.2 and Windows 7). We originally also considered network adapter latencies but after encountering difficulty with this model, chose to effectively include them in the network stack latency. This yields 7 variables with different combinations along each of our measured paths. With the router-originated path measurements discarded, we perform a least-squares linear regression on our 21 remaining path combinations and the *minimum* latency encountered over our latency tests. The results of the regression are included in Table 5.3. Note that all of these latencies are one-way except the router forwarding latency, which approximates both input and output. For the purposes of this chart, we assume that one-way latency is approximately half the round-trip latency.

While these are only an approximation, it is particularly interesting to note that the Windows VM Bridge seems to perform consistently faster than that on Linux. This correlation was first noticed in our VM self-ping measurements (see Table 5.2) but has consistently appeared in every test. While average latencies did not show such a drastic (935%) difference, we still see a notable difference of 397% based on VM self-ping averages. There was not a statistically-significant difference in the self-ping time of Vanilla vs XOR-enhanced FreeBSD virtual machines running on the same host.

Variable	Estimated Minimum (ms)
Ethernet Link	0.09808
Router Forwarding	0.02732
BSD Network Stack	0.02830
Linux 3.2 Network Stack	0.01341
Linux 2.6 Network Stack	0.05726
Linux VM Bridge	0.08735
Windows VM Bridge	0.00934

Table 5.3: Estimates of Minimum Network Latencies (Mean Square Error 0.008441)

5.2.2 Additional Physical Model Considerations

While our UDP Flooder can send at a maximum rate of about 92 Mbps, this is not always sufficient to cause heavy congestion, particularly for short-lived TCP flows. The Maximum Transmission Unit supported by our network is 1500 bytes. To achieve a better model of router congestion, our flooder sends UDP packets with 1500 byte payloads. When the UDP and IP headers are added, the sent packets exceed the MTU size and undergo IP segmentation at the first router. Since the router must perform additional per-packet processing, this forces a consistent congestion bottleneck for our testing purposes.

While we originally planned to cause IP segmentation for TCP packets as well for the purpose of fair analysis, FreeBSD's TCP implementation uses MTU discovery [40] [27] and thus adjusts packet sizes to avoid segmentation within at most a few RTTs. Tcpdumps verify that we actually send 1448 byte packets. This leads to some complications in our data analysis but should not compromise the integrity of our results. For instance, netstat still records reception of the two pieces of a segmented UDP datagram as a single UDP datagram but loss of either piece (and thus failed reconstruction) will result in a perceived packet loss in UDP statistics. Fortunately, netstat *does* record statistics on IP fragmentation including fragments received, dropped, and reassembled.

5.3 Testing Code

For the purpose of protocol evaluation, we implemented a UDP Flooder, a UDP Listener, a TCP Server, and a TCP Client in C using BSD sockets. The

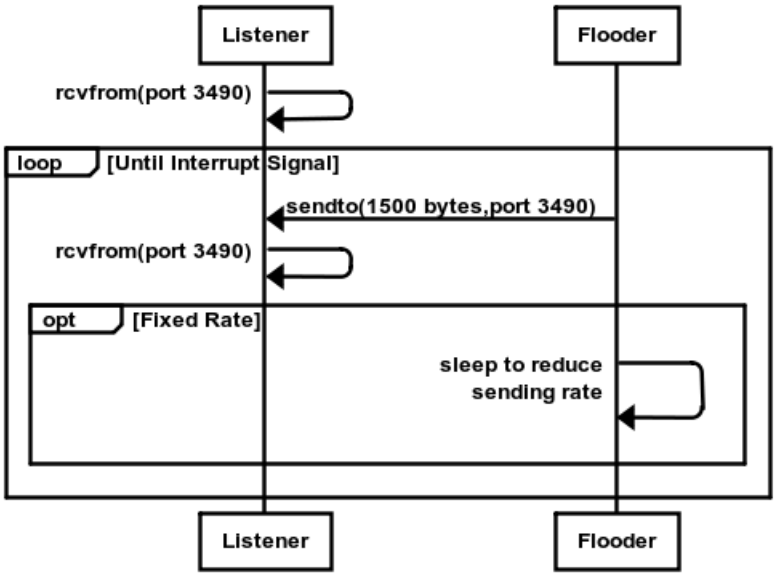

Figure 5.3: Basic callflow of UDP Flooder and Listener binaries

basic callflows of the Flooder-Listener and Server-Client pairs are detailed in Figures 5.3 and 5.4 respectively.

Our UDP Flooder simply sends data as fast as possible unless provided with a hard limit on sending rate. In this case, it sleeps between sending packets to maintain an average sending rate of no more than the hard limit. For the purpose of loss-analysis by the Listener, the first four bytes of each UDP packet sent contain an unsigned 32-bit integer representing the current UDP packet number.

Our UDP Listener constantly receives data and keeps track of the packet numbers received. If a received packet number is larger than the next expected packet number, the loss count is incremented by the difference of $received - expected$. If a packet arrives late but within a 31-bit window of the next expected packet, the loss count is decremented by one.

While our TCP Server can handle an arbitrary number of client connections, we limit it in our testing to one since our virtual machines are restricted to one processor core. The Server binary provides a byte stream of the size requested to any connected Client. Like our UDP Flooder, the first four bytes of each perceived "packet" (defaulting to 1500 bytes but variable via command line) contain the current packet number.

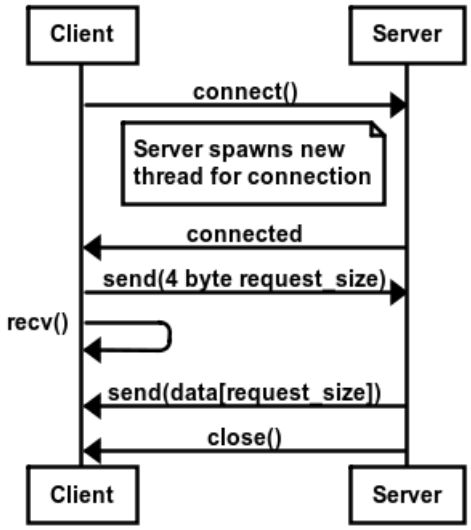

Figure 5.4: Basic callflow of TCP Server and Client binaries

Our TCP Client connects to the Server and sends a 4 byte *request_size*. It then receives the resulting stream and checks that each non-zero 4 byte block of received data corresponds to the next incremental packet number sent by the Server. Since TCP always reliably delivers data in-order to the socket layer, packet number checking provides a TCP assertion mechanism that can allow us to detect improperly reconstructed data from bad XOR-packets. As might be expected, this mechanism proved essential in debugging our implementation.

Simulation behavior is defined through simple built-in s3fnet simulated applications built on top of the simulator's socket framework. These simple applications have approximately equivalent function to the binaries generated for used on our physical network, excepting that the simulation's request size for TCP streams is the first 32 bits of an optionally longer request buffer. The simulation TCP Server also does not do the TCP assertion checking mentioned for our Server binary.

CHAPTER 6

RESULTS AND DISCUSSION

Throughout this chapter, we establish the term "Vanilla" to reference the original BSD TCP implementation as opposed to that modified with our XOR-Packets enhancement. We briefly explore the relation between our protocol enhancement and link latencies under heavy congestion via simulation in S3FNet in Section 6.1. We also verify that S3FNet's behavior is consistent with that of our physical network in Section 6.2. In Section 6.3, we explore the behavior of our protocol for varied filesizes under several congestion loads caused by our UDP Flooder on our physical testbed. Finally, in Section 6.4 we directly compare the performance of our XOR-packets implementation to Vanilla TCP NewReno with and without Selective Acknowledgements [8] (SACKs) enabled. Performance is quantified primarily via the "goodput" of the protocol or the amount of actual non-redundant data transferred per unit time.

6.1 Effect of Link Latency on Throughput

Simulation revealed little difference in goodput between Vanilla TCP and our XOR-Packets enhancement with respect to link latencies. Figure 6.1 shows a slight improvement in goodput for one-way link latencies of 30 ms and 300 ms, however these are limited to improvements about 10.75% and 9.76% respectively. The significant drop in goodput from 30 ms to 300 ms is indicative of the point where link propagation times begin to dominate the entire transmission latency.

At the bottleneck link, it takes 187.5 ms to transmit a maximum-sized 1500 byte packet vs a 100 ms link latency. This effect is compounded when a packet is dropped and we incur TCP's retransmission penalty. Our XOR-packets enhancement outperforms Vanilla TCP in this latency range by avoiding this

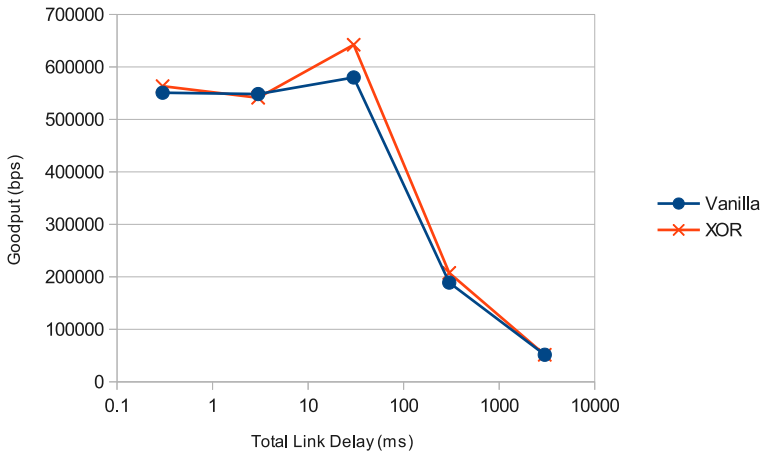

Figure 6.1: Simulated goodput of XOR-Packets vs Regular (Vanilla) TCP for a 2MB download under heavy network congestion

penalty in most cases, despite the 50% additional payload overhead. By the 3 second delay link, however, the inabilty to scale TCP's send window has caused the TCP goodput of both schemes to drop such that a 50% payload overhead is no longer acceptable.

Figure 6.2 shows that our simulated XOR-packets implementation does not perform significantly better than Vanilla TCP with respect to fairness against UDP streams. This makes sense, as our enhancement still has all of TCP's congestion control mechanisms while UDP does not. The slight improvements over Vanilla TCP visible in Figure 6.2 are due entirely to the increased goodput of our enhancements for the 30 ms and 300 ms links already discussed.

The fairness metric as a whole is unusually high for both Vanilla TCP and our enhancement. This is believed to be attributable to the bursty stream interleaving behavior of S3FNet which results in link throughput often being dominated by router buffer size instead of link bandwidth, especially for smaller transfers. Since the router does not perform congestion control on either stream and simply drops TCP and UDP packets equally when its buffer is full, we get a more even goodput distribution than we might otherwise see in real networks as UDP packets arriving after a TCP window burst tend to be dropped while the TCP packets sit in the buffer.

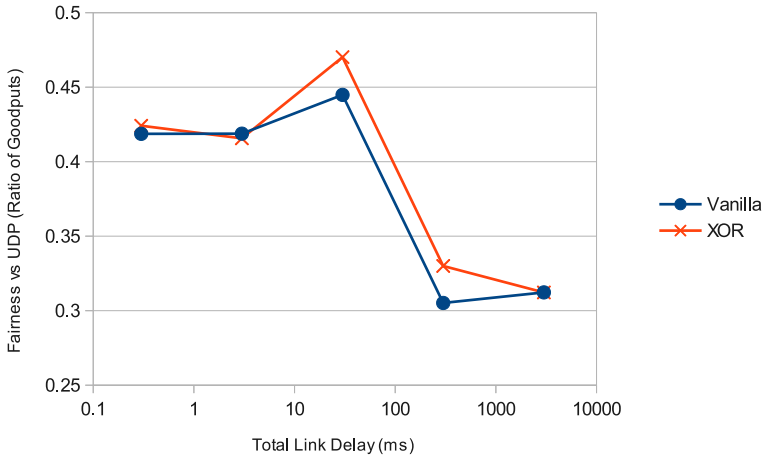

Figure 6.2: Relative fairness of bandwidth usage: Ratio of TCP goodput to UDP goodput

6.2 Simulating our Physical Network

To help verify the behavior of our simulation framework, we attempt to simulate one of the test cases for our physical test network as closely as possible. In particular, we model the transfer a 103500 byte file under heavy congestion using TCP NewReno with our XOR-packets enhancement. With our UDP Flooder sending at a rate of about 77 Mbps (76945272 bps), our TCP transmission completes in 4.516 seconds with a TCP goodput of 183348 bps. This transmission time and goodput agrees within 26.36% of our measured result for transfer time of a 103500 byte file (3.574 seconds) and our measured goodput value of 231693 bps.

The fairly significant deviation from our measured value is not entirely unexpected. We could not accurately model the delays and complexities intrinsic to VirtualBox's network bridge, and we are unsure of the precise buffer size of the E1000 routers used in our physical testbed. We attempted to tune buffer sizes to reflect packet loss rates similar to (but still less than) what we encountered in physical testing. As with our physical Flooder, we had difficulty getting the simulated UDP Flooder to saturate the link but were unable to cause a higher loss rate by forcing IP fragmentation in simulation due to the nature of the simulator's UDP server application. As a result, the Flooder had a slightly higher measured send rate than our network and

resulted in a lower TCP goodput, but even with our adjustments to router buffer size, the simulated packet drop rate was lower than in our physical network.

6.3 Behavior of XOR-Packets Under Varied Congestion

In this section we examine our enhancement on the physical testbed described in Section 5.2 and illustrated in Figure 5.2. We first establish our congestion and packet loss models. We then establish a baseline by looking at Vanilla TCP's performance and comparing it to a known useful enhancement in the form of TCP Selective Acknowledgments. Finally, we examine our XOR-Packets enhancement in detail and consider our XOR-Packets enhancement combined with Selective Acknowledgements.

We specifically consider three levels of congestion. "No congestion" simply means the absence of any UDP flow. "Some congestion" means rate-limited to 50 Mbps, or about half the link bandwidth. Rate-limiting is done via comparison of the limit to our average sending rate in the Flooder code and is a hard limit after an initial burst of up to 100 KB. "High congestion" refers to our UDP Flooder's maximum possible sending rate. For our IP fragmentation model, this is about 75 Mbps but causes almost full router resource utilization.

Goodputs were measured via the binaries described in Section 5.3 using transmitted stream size and flow completion time as measured via the *gettimeofday*() C library call. Measurement was performed client-side and began when a connection was established. Measurement was terminated on reception of the final byte of the requested stream. We performed testing for TCP streams of sizes 1 KB, 10 KB, 1 MB, 10 MB, and 100 MB and UDP rate limits of 0, 50, and 100 Mbps for each of four protocol implementations ("Vanilla" TCP, XOR-Packets, TCP-SACK, and as an additional experiment XOR-SACK). For all completion times under a minute, 10 trials were performed. For tests under 1000 seconds, 5 trials were performed. Finally, for tests taking over 1000s per successful run, we ran three trials. A table with our full aggregated test data including number of trials, mean goodput, and standard deviation for each testing scenario is included as Appendix A.

6.3.1 UDP Congestion Model

As noted in Section 5.2.2, we use IP fragmentation to achieve higher packet loss rates that would conventionally be achievable with a single Flooder by causing additional router processing and thus further congestion at the network bottleneck. Figure 6.3 details UDP packet loss rates a function of sending rate both for normal UDP and for the IP fragmentation model used in our tests.

From Figure 6.3, it is clear that IP segmentation provides a drastic boost in our network's packet loss rate. To achieve such a loss rate otherwise, we would have needed multiple Flooder machines. This might have created unrealistically high congestion rates through extreme link oversubscription. With a single Flooder, control and statistics are simplified, and we can model a significant loss rate without fully utilizing the link in question.

Of course, our fragmentation-based congestion model is not without its drawbacks. As TCP flows are introduced to the network and TCP throughput increases, we see a fluctuation in UDP sending rate. This is primarily attributable to the fact that our E1000 consumer routers put any hosts not connection through their single WAN port on an Ethernet LAN. As per the IEEE 802.3 standard [41], Ethernet links use CSMA/CD in the form of a "line busy" signal. Since our TCP server and UDP Flooder are technically on the same LAN, despite separate physical cables, we often see a drop in sending rate, particularly for the Flooder. Additionally, the interleaving of TCP packets into the router's slightly reduces the average work done by the router since there is no IP fragmentation. As a consequence, we measure slightly reduced loss rates from those expected for UDP flooding only. As drop rates only decrease by at most 5% of their original rate, this does not impede our testing. It *does*, however, make analysis of actual UDP throughput challenging.

Both Selective Acknowledgements and our XOR-Packets enhancement rely on a limited number of packet losses within a fixed sequence number block to function effectively. XOR-packet reconstruction, particularly if using Simple Cache Selection (see Section 3.1.2), specifically relies on the arrival of consecutive packets. While we can theoretically lose up to one third of the entire packet stream, losing any three consecutive packets guarantees the need for a retransmission. Losing certain combinations of two consective packets is

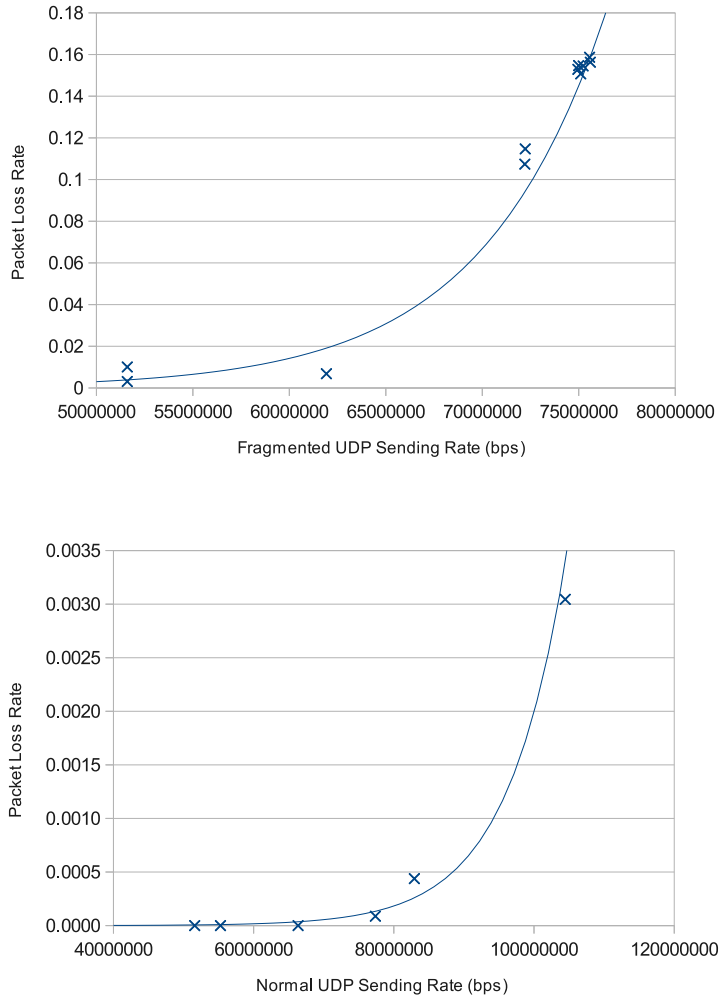

Figure 6.3: Packet loss rate vs UDP sending rate for IP-fragmented UDP [top] and regular UDP [bottom]

also problematic, particularly losing both source packets for an XOR-packet or the second source packet and the XOR-packet itself. With this in mind, suppose that we knew the distribution of consecutive packet drops such that we could reference $P_{seq}(N)$ as the probability of being part of a consective N-packet drop, given that we already know a packet is part of a drop. We could then define the probability of requiring a retransmission given a loss rate R_{loss} as

$$P\left(restransmit\right)_{XOR} = R_{loss}\left(\frac{2}{3}P_{seq}\left(2\right) + \sum_{i=3}^{\infty} P_{seq}\left(i\right)\right). \tag{6.1}$$

Each packet has an R_{loss} chance of being dropped and thus being part of a drop sequence. We then have a $\frac{2}{3}$ chance of needing to restransmit if 2 packets are dropped and *must* retransmit if more than two packets are dropped. Figure 6.4 shows the distribution of consecutive drops for an arbitrary fragmented UDP stream. We consider this to provide an upper-bound drop model for both TCP and UDP streams in our high-congestion test cases as it provides an approximate $P_{seq}(N)$. Substituting our approximate values into Equation 6.1 yields

$$P\left(restransmit\right)_{XOR} = R_{loss}\left(\frac{2}{3}\left(.2433\right) + .5627\right) \approx .7249R_{loss}. \tag{6.2}$$

For comparison,

$$P\left(restransmit\right)_{Vanilla-TCP} = R_{loss}. \tag{6.3}$$

6.3.2 Vanilla TCP Behavior

We first examine Vanilla TCP to establish a baseline for protocol behavior at various congestion levels. Figure 6.5 shows Vanilla TCP performance in the presence of several levels of congestion. In the absence of congestion, Vanilla TCP scales up exponentially as expected for transfers of up through 1 MB. However, its goodput then abrubtly tails off as TCP enters the congestion-avoidance state

A clear peak is visible at around 1 MB for light congestion. This occurs

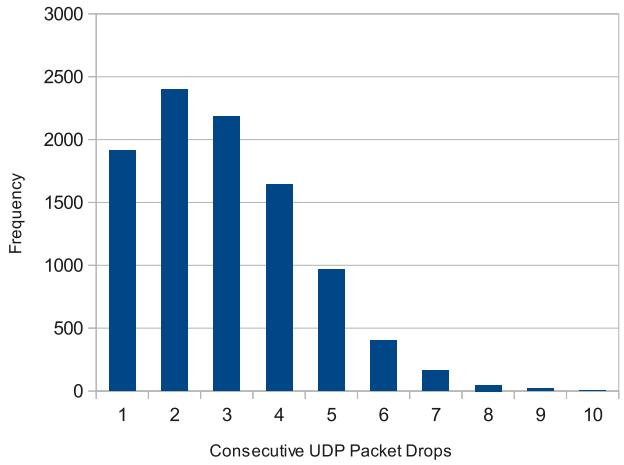

Figure 6.4: Distribution of consecutive packet drops measured for IP-fragmented UDP

for two reasons. First, our Client's TCP receive window fills up entirely and the Sender must consequently delay until a TCP window update arrives. FreeBSD implements congestion window scaling [42] and establishes a maximum window size of 4MB (65536 << *winscale* bytes, where *winscale* = 6) during the initial TCP handshake (FreeBSD's maximum send and receive buffers are 2 MB). It then uses a starting congestion window of 65536 bytes. Our flows of 1 KB and 10KB could never fill the receive window, and in the presence of TCP's slow-start algorithm, our flow of 100 KB also does not fill the receive window. At 1 MB, TCP is able to fill the receive window before the transfer completes. This leads to an idle period where TCP must wait for an acknowledgement or a TCP window update that advertises free or increased receive window space.

This update is generally processed after TCP's minimum allowed timer expires (with exponential backoff if an ACK has not yet arrived), but herein lies our second problem. Our test network's RTT of 1 to 1.5 ms is less than FreeBSD's global minimum timer, which assumes a 30 ms granularity. While this is a reasonable assumption in most real networks, it is potentially up to 30 RTTs in ours! This generates an abnormally long delay in our sending rate any time the receive window fills up and badly hurts our throughput, particularly for the smallest stream sizes that trigger this condition (1 MB

in our tests).

For stream sizes above 1 MB, TCP is able to scale the receiver window such that this sender-side idle time occurs less frequently as time increases. Larger files, such as our 10 MB and 100 MB streams, also average out the lower throughput from TCP's slow-start phase over their longer transfer times.

6.3.3 Vanilla Congestion Behavior

When we introduce some congestion, we see an immediate drop in throughput, especially for our largest stream sizes. This is due primarily to light packet loss. Again, the local goodput maximum around 1 MB is directly attributable to TCP buffer sizes and our network's abnormally low latency. For our 1 MB stream, congestion *does* decrease throughput, but this, along with along with a slightly higher latency from queueing delay, prevents us from exhausting the entire receive buffer. As such, we never incur the 30 ms TCP timer penalty that killed throughput in a congestionless environment, and our overall goodput is not harmed until we begin losing more packets.

Under high congestion, our stream behavior can vary widely depending on the amount and distribution of packet losses. This uncertainty is represented by the large error bar around 100 KB in Figure 6.5's high congestion plot. In general, large error bars in our goodput analysis correspond to points of high potential packet loss. At these points, goodput either continues to significantly increase (in the absence of loss) or drops substantially. This results in a wide variance of goodputs and thus a large confidence-interval range. Here, we see that goodput is drastically reduced as TCP is forced to retransmit packets, but smaller (and thus more short-lived) streams have a better chance of avoiding loss. As stream size (and consequently duration) increases, all streams find a comparable average goodput as slow-start either quickly shifts to TCP's congestion-avoidance state or TCP experiences a timeout.

6.3.4 Vanilla SACK Behavior

TCP Selective Acknowledgement is established as a useful enhancment with meaningful performance improvements in congested environments [8]. Most

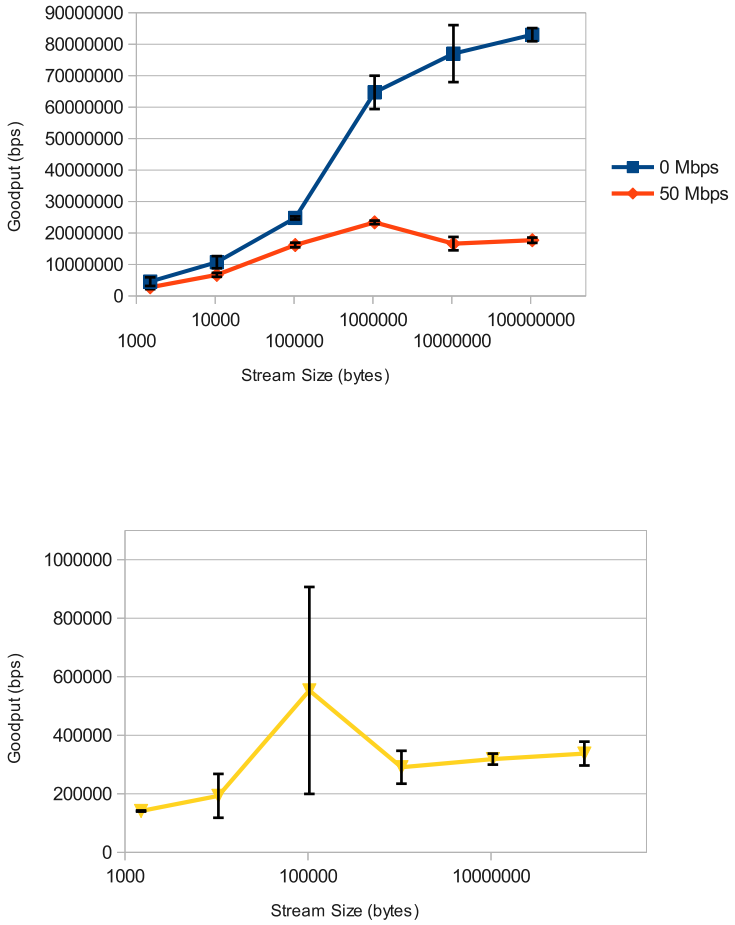

Figure 6.5: Goodput of Vanilla TCP by transfer size for no/low congestion [top] and high (non-rate-limited) congestion [bottom]. Error bars represent a 95% confidence interval. The local maxima seen in the congestion-affected streams are due primarily to implementation-defined buffer sizes and our testbed's very low latencies.

notably it reduces retransmission overhead for multiple discontinuous packet losses and thus serves as a useful model with which to compare or potentially combine our XOR-packets enhancement, which aims to avoid retransmission altogether.

Under low or no congestion, Figure 6.6 shows that SACK-enabled TCP (hereafter referred to as "TCP-SACK") achieves goodput almost identical to that of Vanilla TCP as stream size increases. As we near link saturation, we do see a plateau in goodput for the largest stream sizes where the slight overhead of the SACK-scheme's TCP header extension becomes apparent. However SACK's avoidance of unnecessary retransmissions more than makes up for this penalty in congested conditions. We can see that while goodput is still significantly reduced in low congestion, goodput values degrade from the 1 MB local maximum somewhat less abruptly than with Vanilla TCP. Under high congestion, SACK's effects are illustrated by the additional large error bar around 1 MB. This bar represents the potential for identification and retransmission of lost packets without unnecessary redundant retransmissions, thus allowing for a slower degradation of goodput than in Vanilla TCP. In a sense, the uncertainty we saw in TCP's congestion behavior is extended to larger streams as our retransmission overhead is reduced, thus increasing the chance of higher goodputs.

6.3.5 XOR Congestion Behavior

Since one in three packets in any stream is redundant data, we cannot achieve peak goodputs beyond $\frac{2}{3}$ of the link capacity or about 66 Mbps in this case. This would suggest that in congestionless environments, XOR streams should achiveve peak throughput via slow-start and then abruptly level off. In Figure 6.7, we actually see the overhead of XOR in the reduced ramp-up of goodput compared to Vanilla TCP. Our enhancement's goodput increase loses its concavity for a much smaller stream size and progresses fairly linearly afterwards. Furthermore, we can see that XOR-Packets achieves a much lower maximum goodput than Vanilla TCP and TCP sack, as we would expect from its added overhead.

Figure 6.8 shows that under high congestion, our enhancement's goodput continues to scale with stream size although with diminishing increases For

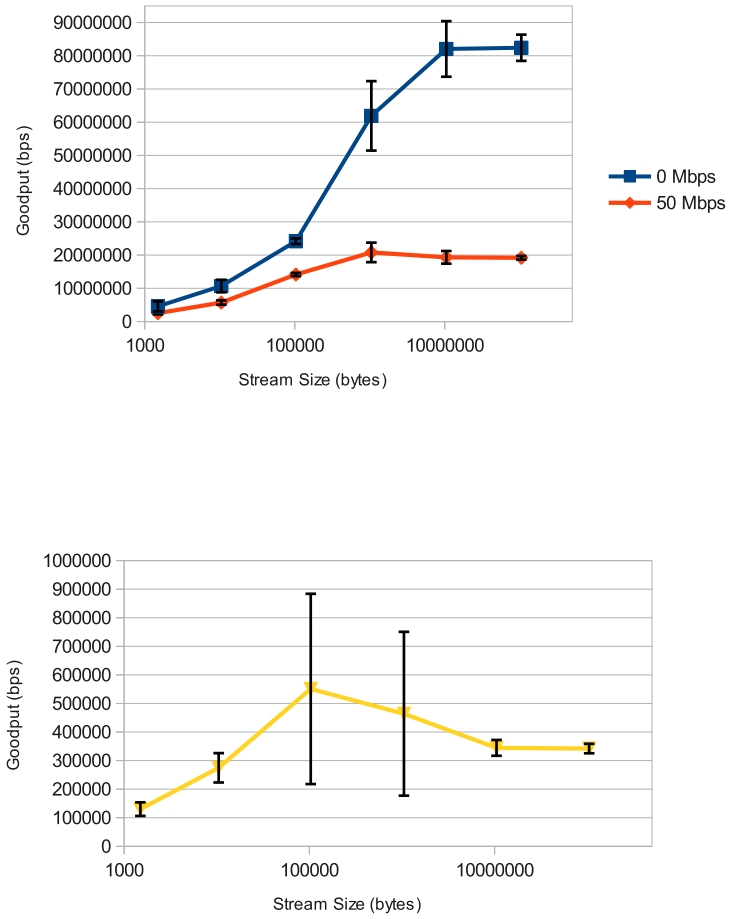

Figure 6.6: Goodput of SACK-enabled Vanilla TCP by transfer size for no/low congestion [top] and high (non-rate-limited) congestion [bottom]. Error bars represent a 95% confidence interval.

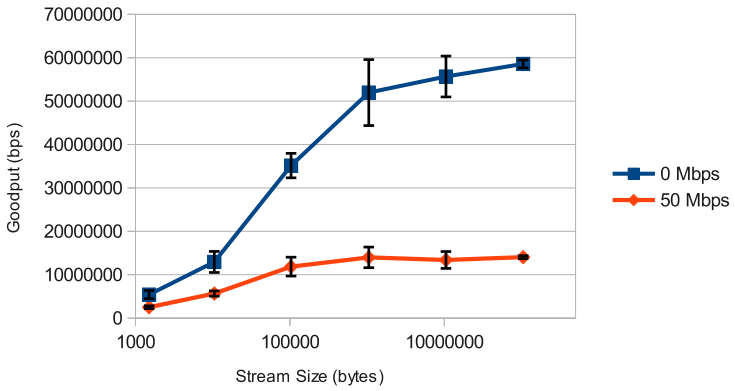

Figure 6.7: Goodput of TCP with XOR-Packets by transfer size for no and low congestion levels. Error bars represent a 95% confidence interval.

smaller streams, the overall goodput is drastically reduced due to the extra overhead. The latency improvements from avoiding retransmission could potentially still be substantial, but in our extremely low-latency network, there is almost no conceivable gain for XOR streams of less than 1 MB. However, as we increase stream size, we see that our goodput still appears to be somewhat scalable, a surprising property given the link utilization under high congestion. This scalability appears sublinear and should consequently converge to an average value as stream size continues to increase, but even this weak scalability is a property that we *do not* observe for either TCP or TCP-SACK.

Most notably, our confidence intervals stay fairly consistent as stream size (and thus goodput) increases. This directly suggests that we are achieving a degree of congestion-resilience, as there is no point where we consistently begin losing packets. Instead, we have a consistent chance of packet loss mitigated by our ability to reconstruct many potential losses.

6.4 Comparison of Protocol Performance

Figure 6.9 exhibits what we have already established: in the absence of congestion, our 50% payload overhead is a pure performance penalty with respect

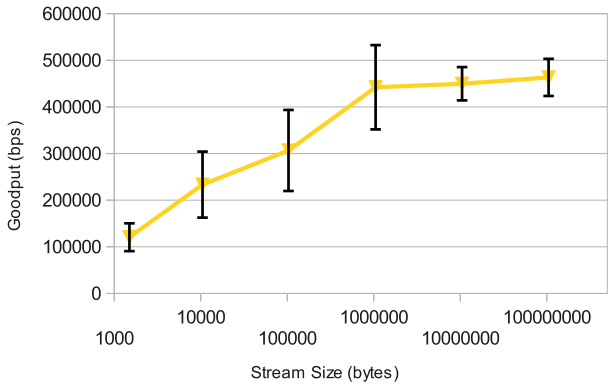

Figure 6.8: Goodput of our XOR-Packets enhancement under high congestion levels, isolated for increased detail. Error bars represent a 95% confidence interval. Unlike Vanilla TCP and TCP-SACK, goodput continues to increase slowly even for high stream sizes.

to goodput, as it cannot scale to the full bandwidth of a link. Any goodput gains must come from the weak scalability of our enhancement in a congested network. However, we do see some interesting behavior. While TCP clearly scales up its window exponentially via slow-start, our enhancment scales up somewhat linearly, actually outperforming Vanilla TCP within our window of error for a brief section of stream size. Given that the XOR-Packets graph loses its concavity early, we predict that TCP may actually be entering the congestion-avoidance state earlier than usual, resulting in an early linear window increase instead of exponential growth. If triggered for a small window, such behavior could provide additional resilience against congestion by giving up the ability to discover unused bandwidth. Thus our added overhead can be advantageous in situations where we expect high congestion as the reduced performance of slow start prevent the slow-start drop problem introduced in Section 2.3.

Figure 6.10 compares the goodput of Vanilla TCP to that of our enhancement in the presence of "Some Congestion". We see that through a stream size of 1 MB, Vanilla TCP has a consistently higher goodput than XOR-packets, but as we increase stream size further, Vanilla TCP goodput drops sharply and must achieve a stable link-sharing rate with the UDP stream

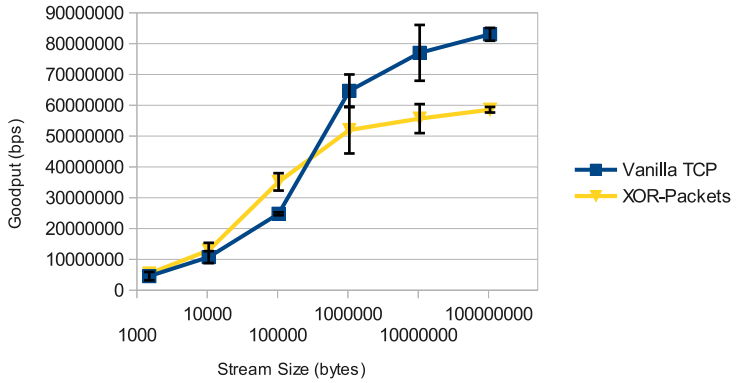

Figure 6.9: Our XOR-Packets enhancement vs Vanilla TCP in the absence of congestion. Error bars represent a 95% confidence interval.

causing congestion. XOR-packets appears to have quickly stabilized as of our 100K stream.

Figure 6.11 shows that in heavy network congestion, Vanilla TCP experiences a significant drop in throughput indicative of potential packet loss. Vanilla TCP appears to then begin to converge towards an average goodput for almost all stream sizes, indicative of link-sharing behavior with other streams. In comparison, our XOR-packets enhancement appears to have goodput that is weakly scalable with stream size, achieving slightly increased goodput even in the congested environment for streams over 1 MB in size.

In heavy congestion, TCP-SACK appears to converge around the same average goodput, again, probably its fair subscription of the link. However TCP-SACK's transition up to and down from the local maximum at 1 MB is much more linear than that of Vanilla TCP. This is presumably since TCP-SACK can prevent unnecessary retransmissions for multiple holes within the TCP window, but as drops and the number of gaps increases, it suffers additional overhead and begins to converge to the same average number of retransmissions as Vanilla TCP.

By comparison, XOR-Packets is significantly more scalable since it maintains no additional state as loss increases. It should eventually converge towards $\frac{2}{3}$ of TCP's convergent goodput (ie, all XOR-packets are lost but

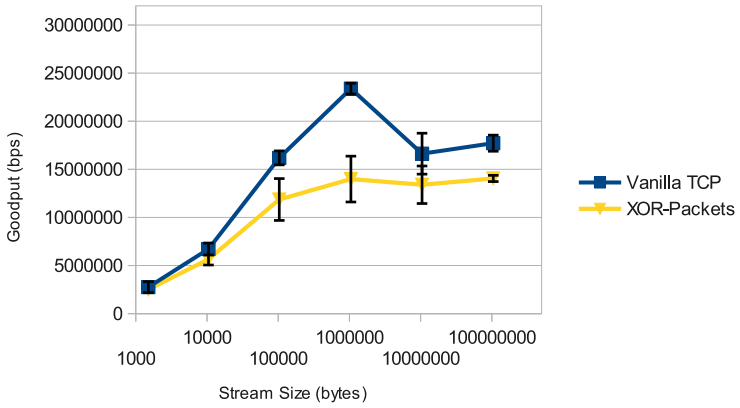

Figure 6.10: Our XOR-Packets enhancement vs Vanilla TCP in the presence of congestion. Error bars represent a 95% confidence interval.

we still had their transmission overhead), but this requires the packet loss rate to approach $\frac{1}{3}$ at which point TCP's congestion window could easily be reduced to near-zero values as a response to excessive loss, killing throughput altogether.

While TCP-SACK's goals are certainly tangential to those of our XOR-packets enhancement, we argue that they are mostly non-overlapping. Selective Acknowledgements work particularly well when sequential groups of packets are dropped or received (ie the number of gaps in the TCP window is minimized). XOR-packets with Simple Cache Selection rely on *not* dropping sequential groups of packets. Furthermore, SACKs aim to limit retransmissions to only those necessary whereas XOR-packets provide a mechanism for avoiding retransmission altogether. This suggests that when paired together, SACKs may act as an additional resilience heuristic to minimize retransmission of any packets that XOR-packets cannot reconstruct.

Our FreeBSD implementation of XOR-packets was written so as to not be explicitly incompatible with SACKs. Unfortunately, for higher stream sizes, there appear to be some SACK-retransmit hooks which we did not properly modify for compatility with XOR-packets. Despite an explicit assertion added to the FreeBSD network stack that XOR-packets may not be retransmitted, the SACK scheme can rarely retransmit an XOR-packet in

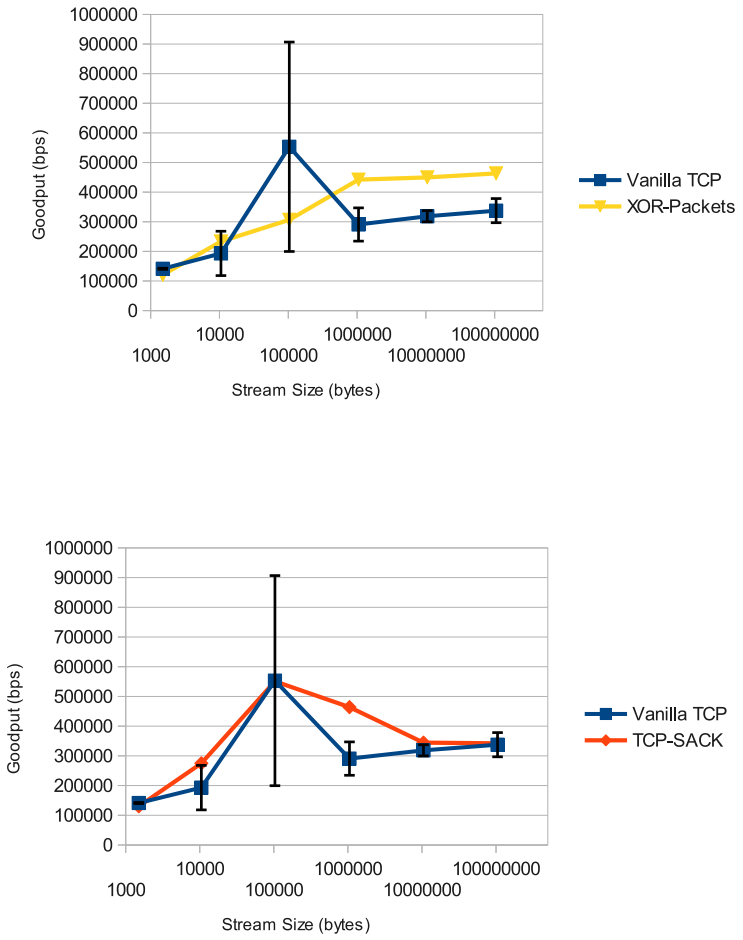

Figure 6.11: Vanilla TCP vs our XOR-Packets enhancement [top] and vs SACK-enabled Vanilla [bottom] under high congestion conditions. Error bars are omitted from TCP-SACK and our XOR-Packets enhancement for increased readability but are the same as those in Figures 6.6, and 6.8.

lieu of the actual data packet requested. SACK also cannot distinguish between the sequence number of an XOR-packet and the sequence number of its second source packet. Suppose an XOR-packet XOR(4/5) is cached and packet 4 is missing. The sack scheme will recognize XOR(4/5) as packet 5 and may rarely retransmit something like XOR(6/4). This gets reconstructed to garbage, violating the correctness of TCP, and the invalid packet 4 gets ACK'd, meaning 4 can never be received and proper reconstruction can never occur. With an advanced packet selection scheme such as Spaced Subwindow Selection (section 3.1.4) that allows XOR construction from two non-consecutive source packets, this should be fixable with minimal modifications to SACK logic.

This deadlock scenario only happens rarely for large packet streams. As such, we were still able to test our XOR-SACK hybrid implementation for 10K, 100K, and 1M stream sizes. The goodput of this limited test case is displayed in Figure 6.12. For each stream size tested the combined XOR-SACK scheme has goodput greater than or equal to that of Vanilla TCP, TCP-SACK, *and* our regular XOR-Packets enhancement. While it looks to converge towards the same final average goodput as our regular XOR-Packets enhancement, we can only speculate that the same relative performance gains will be maintained for streams of 10 MB and greater.

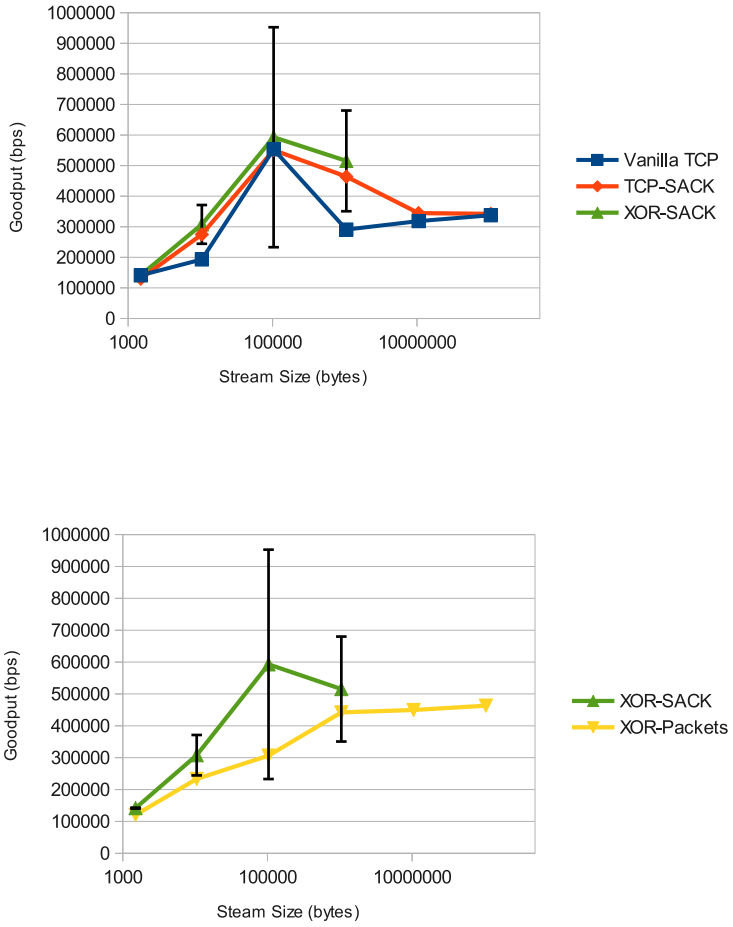

Figure 6.12: Performance of SACK+XOR-enabled TCP under high congestion vs Vanilla implementations [top] and regular XOR [bottom]. Error bars represent a 95% confidence interval for XOR-SACK only. Other error bars are omitted for readability but may be referenced in the other presented figures.

CHAPTER 7

CONCLUSION

While TCP is designed to prevent congestion collapse, it nearly guarantees low levels of congestion via the slow-start mechanism and consequently one or more packet drops in the presence of other flows. We have shown that when TCP drops a packet, there is a delay incurred that, even in the best case, is significant for high latency networks.

We have introduced a packet-loss-tolerant TCP protocol through the use of FEC in the form of XOR-packets. We have established that this scheme is the $N = 2$ case of a more general model of loss tolerance which guarantees the property that we can lose and reconstruct at least one packet per N within a stream of arbitrary length.

We have successfully implemented this protocol as an enhancement to the FreeBSD TCP stack and found that it can result in higher goodput for larger TCP streams in a congested network despite a minimum 50% payload overhead. For stream sizes of 10 MB and more experiencing high congestion levels, the results surpass SACK-enabled NewReno, a widely used and accepted TCP implementation. We also find that our incomplete XOR-packets implemented with SACK performs suprisingly well even for lower stream sizes. Finally for both XOR and SACK-enabled XOR, we see indications of weak goodput scalability as stream size is increased even under high congestion levels, suggesting that our protocol does achieve a low degree of congestion resilience.

Some directions for future research would be an evaluation of the protocol for higher delay links where the TCP retransmission penalty is higher. Given that we achieve a degree of congestion resilience, it would be interesting to test the latency improvements achieved for very short flows that experience sporadic congestion-induced drops, particularly when traversing high delay-bandwidth links.

Given that packet loss no longer guarantees a TCP retransmission, we may

be able to leverage packet loss as a non-binary metric for congestion control or optimized goodput vs congestion resilience. Our choice of N could be made adaptive based on the perceived lossiness of the network. We could also explore the loss-resilience of this enhancement in known lossy environments like wireless.

The problem-space could also be explored via the alternative window designs presented in Section 3.1 which provide better guarantees when faced with bursty packet loss. Like larger N values, these introduce a tradeoff between the worst case latency before XOR reconstruction can occur for a loss and the overhead incurred.

Finally, the issue of fairness (with respect to bandwidth allocation) should be considered, given the additional bandwidth usage of this enhancement. It is unknown how this protocol's bandwidth allocation compares to that of other TCP flows when sharing a link with multiple flows, particularly when those flows belong to other TCP variants. Furthermore, the actual effect on UDP flow loss rates and the resulting fairness and was not directly measurable due to our IP fragmentation approach and fluctuations in the UDP sending rate due to the Ethernet LAN busy signal.

REFERENCES

[1] R. Hack, "The google fiber project," *J. Comput. Sci. Coll.*, vol. 28, no. 5, pp. 140–140, May 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2458569.2458598

[2] V. G. Cerf and R. E. Khan, "A protocol for packet network intercommunication," *IEEE TRANSACTIONS ON COMMUNICATIONS*, vol. 22, pp. 637–648, 1974.

[3] J. Postel, "Transmission Control Protocol," RFC 793 (INTERNET STANDARD), Internet Engineering Task Force, Sep. 1981, updated by RFCs 1122, 3168, 6093, 6528. [Online]. Available: http://www.ietf.org/rfc/rfc793.txt

[4] J. Postel, "User Datagram Protocol," RFC 768 (INTERNET STANDARD), Internet Engineering Task Force, Aug. 1980. [Online]. Available: http://www.ietf.org/rfc/rfc768.txt

[5] V. Jacobson, "Congestion avoidance and control," *SIGCOMM Comput. Commun. Rev.*, vol. 18, no. 4, pp. 314–329, Aug. 1988. [Online]. Available: http://doi.acm.org/10.1145/52325.52356

[6] M. Allman, V. Paxson, and W. Stevens, "TCP Congestion Control," RFC 2581 (Proposed Standard), Internet Engineering Task Force, Apr. 1999, obsoleted by RFC 5681, updated by RFC 3390. [Online]. Available: http://www.ietf.org/rfc/rfc2581.txt

[7] S. Floyd, T. Henderson, and A. Gurtov, "The NewReno Modification to TCP's Fast Recovery Algorithm," RFC 3782 (Proposed Standard), Internet Engineering Task Force, Apr. 2004, obsoleted by RFC 6582. [Online]. Available: http://www.ietf.org/rfc/rfc3782.txt

[8] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP Selective Acknowledgment Options," RFC 2018 (Proposed Standard), Internet Engineering Task Force, Oct. 1996. [Online]. Available: http://www.ietf.org/rfc/rfc2018.txt

[9] K. Fall and S. Floyd, "Simulation-based comparisons of tahoe, reno and sack tcp," *SIGCOMM Comput. Commun. Rev.*, vol. 26, no. 3, pp. 5–21, July 1996. [Online]. Available: http://doi.acm.org/10.1145/235160.235162

[10] C. Casetti, M. Gerla, S. Mascolo, M. Y. Sanadidi, and R. Wang, "Tcp westwood: end-to-end congestion control for wired/wireless networks," *Wirel. Netw.*, vol. 8, no. 5, pp. 467–479, Sep. 2002. [Online]. Available: http://dx.doi.org/10.1023/A:1016590112381

[11] L. A. Grieco and S. Mascolo, "Performance evaluation and comparison of westwood+, new reno, and vegas tcp congestion control," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 25–38, Apr. 2004. [Online]. Available: http://doi.acm.org/10.1145/997150.997155

[12] L. Brakmo and L. Peterson, "Tcp vegas: end to end congestion avoidance on a global internet," *Selected Areas in Communications, IEEE Journal on*, vol. 13, no. 8, pp. 1465–1480, 1995.

[13] J. Sing and B. Soh, "Tcp new vegas: Improving the performance of tcp vegas over high latency links," in *Network Computing and Applications, Fourth IEEE International Symposium on*, 2005, pp. 73–82.

[14] D. X. Wei, C. Jin, S. H. Low, and S. Hegde, "Fast tcp: motivation, architecture, algorithms, performance," *IEEE/ACM Trans. Netw.*, vol. 14, no. 6, pp. 1246–1259, Dec. 2006. [Online]. Available: http://dx.doi.org/10.1109/TNET.2006.886335

[15] S. Liu, T. Başar, and R. Srikant, "Tcp-illinois: a loss and delay-based congestion control algorithm for high-speed networks," in *Proceedings of the 1st international conference on Performance evaluation methodolgies and tools*, ser. valuetools '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1190095.1190166

[16] L. Xu, K. Harfoush, and I. Rhee, "Binary increase congestion control (bic) for fast long-distance networks," in *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, vol. 4, 2004, pp. 2514–2524 vol.4.

[17] S. Ha, I. Rhee, and L. Xu, "Cubic: a new tcp-friendly high-speed tcp variant," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 64–74, July 2008. [Online]. Available: http://doi.acm.org/10.1145/1400097.1400105

[18] S. Floyd, "HighSpeed TCP for Large Congestion Windows," RFC 3649 (Experimental), Internet Engineering Task Force, Dec. 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3649.txt

[19] D. Leith and R. Shorten, "H-tcp: Tcp for high-speed and long-distance networks," in *in Proc. PFLDnet, Argonne*, 2004. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.3.7816

[20] J. Wang, J. Wen, J. Zhang, and Y. Han, "Tcp-fit: An improved tcp congestion control algorithm and its performance." in *INFOCOM*. IEEE, 2011, pp. 2894–2902.

[21] C. Barakat and E. Altman, "Bandwidth tradeoff between {TCP} and link-level {FEC}," *Computer Networks*, vol. 39, no. 2, pp. 133 – 150, 2002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S138912860100305X

[22] D. Vassis, G. Kormentzas, A. Rouskas, and I. Maglogiannis, "The ieee 802.11g standard for high data rate wlans," vol. 19, pp. 21–26, May 2005.

[23] R. Dunaytsev, D. Moltchanov, Y. Koucheryavy, and J. Harju, "Modeling tcp sack performance over wireless channels with completely reliable arq/fec," *International Journal of Communication Systems*, vol. 24, no. 12, pp. 1533–1564, 2011. [Online]. Available: http://dx.doi.org/10.1002/dac.1230

[24] K. Park and W. Wang, "Afec: An adaptive forward error-correction protocol for end-to-end transport of real-time traffic," in *In Proc. IEEE IC3N*, 1997, pp. 196–205.

[25] A. Bestavros and G. Kim, "Exploiting redundancy for timeliness in tcp boston," in *Real-Time Technology and Applications Symposium, 1997. Proceedings., Third IEEE*, 1997, pp. 184–190.

[26] A. Bestavros and G. Kim, "Tcp boston: a fragmentation-tolerant tcp protocol for atm networks," in *INFOCOM '97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution., Proceedings IEEE*, vol. 3, 1997, pp. 1210–1217 vol.3.

[27] J. Mogul and S. Deering, "Path MTU discovery," RFC 1191 (Draft Standard), Internet Engineering Task Force, Nov. 1990. [Online]. Available: http://www.ietf.org/rfc/rfc1191.txt

[28] D. Nicol, D. Jin, and Y. Zheng, "S3f: The scalable simulation framework revisited," in *Simulation Conference (WSC), Proceedings of the 2011 Winter*, 2011, pp. 3283–3294.

[29] D. Jin, Y. Zheng, H. Zhu, D. Nicol, and L. Winterrowd, "Virtual time integration of emulation and parallel simulation," in *Principles of Advanced and Distributed Simulation (PADS), 2012 ACM/IEEE/SCS 26th Workshop on*, 2012, pp. 201–210.

[30] T. F. Foundation, "The freebsd project." [Online]. Available: http://www.freebsd.org/

[31] W. R. Stevens and G. R. Wright, *TCP/IP illustrated (vol. 2): the implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[32] L. Stewart and J. Healy, "Light-weight modular tcp congestion control for freebsd 7," CAIA, Tech. Rep. 071218A, Dec. 2007.

[33] L. Budzisz, R. Stanojevic, R. Shorten, and F. Baker, "A strategy for fair coexistence of loss and delay-based congestion control algorithms," *Communications Letters, IEEE*, vol. 13, no. 7, pp. 555–557, 2009.

[34] D. Hayes and G. Armitage, "Improved coexistence and loss tolerance for delay based tcp congestion control," in *Local Computer Networks (LCN), 2010 IEEE 35th Conference on*, 2010, pp. 24–31.

[35] V. Jacobson, "4bsd header prediction," *ACM Computer Communication Review*, April 1990.

[36] N. Spring, D. Wetherall, and D. Ely, "Robust Explicit Congestion Notification (ECN) Signaling with Nonces," RFC 3540 (Experimental), Internet Engineering Task Force, June 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3540.txt

[37] P. Srisuresh and M. Holdrege, "IP Network Address Translator (NAT) Terminology and Considerations," RFC 2663 (Informational), Internet Engineering Task Force, Aug. 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2663.txt

[38] Tcpdump, "Tcpdump." [Online]. Available: http://www.tcpdump.org/

[39] Oracle, "Oracle vm virtualbox." [Online]. Available: https://www.virtualbox.org/

[40] J. Postel, "The TCP Maximum Segment Size and Related Topics," RFC 879, Internet Engineering Task Force, Nov. 1983, updated by RFC 6691. [Online]. Available: http://www.ietf.org/rfc/rfc879.txt

[41] "Ieee standard for information technology–telecommunications and information exchange between systems–local and metropolitan area networks– specific requirements part 3: Carrier sense multiple access with collision detection (csma/cd) access method and physical layer specifications - section five," *IEEE Std 802.3-2005 (Revision of IEEE Std 802.3-2002 including all approved amendments)*, vol. Section5, pp. 1–417, 2005.

[42] V. Jacobson, R. Braden, and D. Borman, "TCP Extensions for High Performance," RFC 1323 (Proposed Standard), Internet Engineering Task Force, May 1992. [Online]. Available: http://www.ietf.org/rfc/rfc1323.txt

APPENDIX A

TABLE OF GOODPUT MEASUREMENTS

Table A.1: Mean Goodput Measurements and Standard Deviation from Physical Testbed

Variant	**UDP Rate**	**Size**	**Trials**	**Mean**	**Std. Dev.**
Vanilla	0 Mbps	1 KB	10	4565360	2209223
Vanilla	0 Mbps	10 KB	10	10733518	3124599
Vanilla	0 Mbps	100 KB	10	24796571	769225
Vanilla	0 Mbps	1 MB	10	64692035	8553715
Vanilla	0 Mbps	10 MB	10	77010675	14615896
Vanilla	0 Mbps	100 MB	10	83031395	3378930
Vanilla	50 Mbps	1 KB	10	2752651	919249
Vanilla	50 Mbps	10 KB	10	6715231	1004558
Vanilla	50 Mbps	100 KB	10	16189938	1175283
Vanilla	50 Mbps	1 MB	10	23386125	898075
Vanilla	50 Mbps	10 MB	10	16627738	3428162
Vanilla	50 Mbps	100 MB	10	17717259	1353924
Vanilla	100 Mbps	1 KB	10	141209	3054
Vanilla	100 Mbps	10 KB	10	193035	120819
Vanilla	100 Mbps	100 KB	10	553209	570517
Vanilla	100 Mbps	1 MB	10	290743	90653
Vanilla	100 Mbps	10 MB	5	318528	21441
Vanilla	100 Mbps	100 MB	3	337486	36049
XOR	0 Mbps	1 KB	10	5417509	1537537
XOR	0 Mbps	10 KB	10	12939253	3916930
XOR	0 Mbps	100 KB	10	35146437	4545687
XOR	0 Mbps	1 MB	10	51970559	12261486

Continued on next page

Table A.1 – continued from previous page

Variant	UDP Rate	Size	Trials	Mean	Std. Dev.
XOR	0 Mbps	10 MB	10	55674430	7598348
XOR	0 Mbps	100 MB	10	58567598	1471766
XOR	50 Mbps	1 KB	10	2507857	524990
XOR	50 Mbps	10 KB	10	5657344	970988
XOR	50 Mbps	100 KB	10	11861150	3509285
XOR	50 Mbps	1 MB	10	13988226	3842624
XOR	50 Mbps	10 MB	10	13396019	3144159
XOR	50 Mbps	100 MB	10	14048674	524423
XOR	100 Mbps	1 KB	10	120497	48142
XOR	100 Mbps	10 KB	10	233349	114196
XOR	100 Mbps	100 KB	10	306573	140200
XOR	100 Mbps	1 MB	10	442180	145916
XOR	100 Mbps	10 MB	5	449680	40680
XOR	100 Mbps	100 MB	3	463258	35193
SACK	0 Mbps	1 KB	10	4571651	2435437
SACK	0 Mbps	10 KB	10	10685118	3018943
SACK	0 Mbps	100 KB	10	24167112	1365180
SACK	0 Mbps	1 MB	10	61898491	16869622
SACK	0 Mbps	10 MB	10	82040093	13499491
SACK	0 Mbps	100 MB	10	82400944	6369299
SACK	50 Mbps	1 KB	10	2519088	618025
SACK	50 Mbps	10 KB	10	5710815	1067511
SACK	50 Mbps	100 KB	10	14142324	777900
SACK	50 Mbps	1 MB	10	20808347	4738069
SACK	50 Mbps	10 MB	10	19324589	3070206
SACK	50 Mbps	100 MB	10	19191356	885469
SACK	100 Mbps	1 KB	10	130149	38513
SACK	100 Mbps	10 KB	10	274795	82950
SACK	100 Mbps	100 KB	10	550795	537175
SACK	100 Mbps	1 MB	10	463987	462484
SACK	100 Mbps	10 MB	5	344547	31527

Continued on next page

Table A.1 – continued from previous page

Variant	UDP Rate	Size	Trials	Mean	Std. Dev.
SACK	100 Mbps	100 MB	3	342149	14638
XOR-SACK	0 Mbps	1 KB	10	4891640	2395832
XOR-SACK	0 Mbps	10 KB	10	13029669	3041873
XOR-SACK	0 Mbps	100 KB	10	38741437	5385052
XOR-SACK	0 Mbps	1 MB	10	51804456	12099520
XOR-SACK	0 Mbps	10 MB	10	70677352	19202436
XOR-SACK	0 Mbps	100 MB	10	57844336	2621538
XOR-SACK	50 Mbps	1 KB	10	2540402	727950
XOR-SACK	50 Mbps	10 KB	10	4944132	2196499
XOR-SACK	50 Mbps	100 KB	10	13022752	5403067
XOR-SACK	50 Mbps	1 MB	10	13077291	5332949
XOR-SACK	50 Mbps	10 MB	10	13706006	1616615
XOR-SACK	50 Mbps	100 MB	10	14756456	526926
XOR-SACK	100 Mbps	1 KB	10	141705	1879
XOR-SACK	100 Mbps	10 KB	10	307795	77848
XOR-SACK	100 Mbps	100 KB	10	592861	441758
XOR-SACK	100 Mbps	1 MB	5	515346	142996